AFRL-IF-RS-TR-2000-151
Final Technical Report
October 2000

PROCESSOR-IN-MEMORY APPLICATIONS ASSESSMENT

SM&A Corporation

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. G435

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2000-151 has been reviewed and is approved for publication.

APPROVED: Christopher J. Flynn

CHRISTOPHER J. FLYNN
Project Engineer

FOR THE DIRECTOR: Northrup Fowler

NORTHRUP FOWLER, Technical Advisor
Information Technology Division
Information Directorate

PROCESSOR-IN-MEMORY APPLICATIONS ASSESSMENT

Joseph F. Musmanno,
Joseph W. Manke, and
Jon W. Harris

Contractor: SM&A Corporation
Contract Number: F30602-97-D-0070/Task 0003
Effective Date of Contract: 27 May 1998
Contract Expiration Date: 27 May 1999
Short Title of Work: Processor-In-Memory Applications Assessment
Period of Work Covered: May 98 - May 99

Principal Investigator: Joseph F. Musmanno
Phone: (781) 890-4200
AFRL Project Engineer: Christopher J. Flynn
Phone: (315) 330-3249

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Christopher J. Flynn, AFRL/IFTC, 26 Electronic Pky, Rome, NY.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	OCTOBER 2000	Final May 98 - May 99

4. TITLE AND SUBTITLE PROCESSOR-IN-MEMORY APPLICATIONS ASSESSMENT	5. FUNDING NUMBERS C - F30602-97-D-0070/Task 0003 PE - 62301E PR - 4384 TA - QF WU - 01
6. AUTHOR(S) Joseph F. Musmanno, Joseph W. Manke, and Jon W. Harris	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) SM&A Corporation 1300-B Floyd Avenue Rome New York 13440-4600	8. PERFORMING ORGANIZATION REPORT NUMBER SM&A Report #0NY-1801
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 3701 North Fairfax Drive Arlington VA 22204-1714 Air Force Research Laboratory/IFTC 26 Electronic Pky Rome NY 13441-4514	10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2000-151

11. SUPPLEMENTARY NOTES
Air Force Research Laboratory Project Engineer: Christopher J. Flynn/IFTC/(315) 330-3249

12a. DISTRIBUTION AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	

13. ABSTRACT *(Maximum 200 words)*

This report presents a suite of application-oriented benchmarks, and the methodology supporting it. This document describes the development of a benchmark suite that can be used to quantify the performance gains likely to be achieved for defense computer programs when implemented using approaches and architecture developed under DARPA's Data Intensive Systems (DIS) program.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Data Intensive Systems (DIS), Benchmark, Method of Moments (MoM), Synthetic Aperture Radar (SAR), Fast Multiple Method (FMM)	332
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Standard Form 298 (Rev. 2-89) (EG)
Prescribed by ANSI Std. 239.18
Designed using Perform Pro, WHS/DIOR, Oct 94

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

LIST OF TABLES

1. INTRODUCTION

As part of the DARPA Information Technology Office's Data-Intensive Systems research program, this document presents a suite of application-oriented benchmarks, and the methodology supporting it.

This section provides an introduction to the effort and the document, including an outline of the motivation for the program, the goals to be sought, and the organization of the remaining sections of the document.

1.1 DATA-INTENSIVE SYSTEMS

Many defense applications employ large data sets that are accessed non-contiguously. These applications cannot take full advantage of typical memory-access optimizations, and consequently perform at approximately two orders of magnitude below peak rates. Some data-starved applications identified by DARPA/ITO are RADAR cross-section modeling, high-definition imaging, terrain masking, relational and object-oriented databases, structural dynamics calculations, and circuit simulation.

Compounding the problem, memory access speeds have also not grown in pace with storage sizes, nor with processor speeds.

To bolster the above applications and address these problems, DARPA/ITO has launched a Data-Intensive Systems (DIS) effort, which includes two complimentary tasks: (1) incorporate logic within memory chips (processor-in-memory, or PIM), allowing manipulation of data locally in a memory subsystem; and (2) adaptive cache management, to increase cache utilization and improve data flow.

1.2 MOTIVATION

The development of new architectures and approaches to data-intensive computing could be beneficial to many problems of interest to DARPA. Evaluation of the architectures in the context of those problems is essential in order to realize those benefits.

Equally important, the existence of simplified–but meaningful–programs derived from defense applications can provide valuable input to the development process.

Therefore, benchmarking fills a critical need in the development of Data-Intensive Systems. An appropriate benchmarking effort will accelerate insertion of DIS technology into defense systems.

1.3 GOALS

The primary goal of this effort is the development of a benchmark suite that can be used to quantify the performance gains likely to be achieved for defense computer programs when implemented using approaches and architectures developed under the DIS program.

Any benchmark specification dealing with early research into new systems must remain architecture-neutral. In support of this goal, the benchmark specifications are essentially only the mathematical description of problems' solutions. Of course, over years of development in the context of Von Neumann computer architectures, many known optimizations have been utilized, and an attempt has been made to provide or reference these, so that participants charged with implementing the benchmarks are not faced with having to independently rediscover the optimizations.

Benchmarks that focus on the measurement of relative performance frequently involve implementation only of specific, isolated functions, resulting in accurate measurement of peak performance. This level of performance is rarely realizable in general application, so benchmarks that include the processes of data movement and preparation are desirable for a more generalized measurement of real performance. Considering the variety of architectures under scrutiny in the DIS program, it would be dangerous to presume that these "overhead" functions diminish in proportional resource consumption as data sets grow larger. Therefore, avoidance of isolated tasks as benchmarks is a goal of this program; rather, performance related to the interactions between program components is intended to be included in the measurements.

[Weems], while reviewing lessons from prior benchmark efforts, points out:

"Having a known, correct solution for a benchmark is essential, since it is difficult to compare the performance of architectures that produce different results. For example, suppose architecture A performs a task in half the time of B, but A uses integer arithmetic while B uses floating-point, and they obtain different results. Is A really twice as powerful as B?"

Therefore, a complete solution with test data sets is considered one of the essential components of the distribution of the benchmark specification.

Although there are sometimes competing ideas about how to best solve a particular problem, the goal of a benchmark is not specifically to solve a problem, but rather to test the performance of different machines doing comparable work. Since DIS architectures are likely to vary greatly, significant latitude is allowed in the implementation of a solution to benchmark problems. However, participants must remain cognizant of the fact that ultimately, the measurements taken must be meaningful in the context of defense problems, and specifically in the context of *relative* gain. So, it is not a goal of this benchmark effort to develop the best solutions for the most difficult problems; rather, it is a goal to employ pertinent solutions to problems expected to benefit from DIS research, and allow enough flexibility to maximize individual performance, yet remain consistent and comparable.

While benchmarks that are too simplistic do not offer valuable results, those that are too complex are never implemented, at least in a meaningful way. Resources are limited, so ease of implementation is a factor of consideration. It is a goal of this program to develop benchmark programs that should require relatively little source code during implementation, yet still offer meaningful results.

Often, high-performance systems are developed that remain under-utilized due to the esoteric or difficult nature of their programming. Therefore, an important goal of this effort is to evaluate the labor costs associated with use of candidate architectures. The ability to handle existing, 'legacy code' is an important consideration, as is the labor cost to exploit the powerful features of these systems.

A program will generally execute faster when its required data set is small enough to fit in main memory, as opposed to when paging or swapping is required. Likewise, when the data set is small enough to fit in cached memory, it will generally execute faster still. Balancing the competing factors of speed, size, and cost is a major engineering decision, and quantifying the effects of that decision is a goal of this effort.

Finally, in support of the primary goal of being able to quantify performance gains, it is a goal of this effort to remain open to any additional information participants wish to supply that will assist reviewers in making an accurate determination. While this document specifies minimum participation requirements, information such as results, analyses, proofs, or additional metrics is hereby solicited.

1.4 ORGANIZATION OF THIS DOCUMENT

The remainder of this document is organized as follows:

Section 2 provides the foundation for this work, including analyses of the algorithms included in the benchmark.

Section 3 outlines the procedures to be followed by participants.

Section 4 provides the specifications for the benchmark set.

Sections 4 and 5 give contact information and references, for participants needing additional information.

2. BACKGROUND

Given the motivation and goals outlined above, this section addresses the determination of the content of the benchmark suite. The selection of the algorithms to be included, and analyses of each–showing critical performance bottlenecks and suggesting possible gains–are provided.

2.1 ALGORITHM SELECTION

Although many classes of algorithms could benefit from systems with advanced memory or PIM elements, three classes would provide a representative scope of achievable performance improvement for problems of interest to key DARPA programs:

Model-Based Image Generation – This class includes generation of synthetic signatures and scenes for targets and terrain based on complex models of objects and sophisticated camera models for various sensor types. Applications include target recognition, real-time scene simulation for visualization or training, and model-driven change detection.

Target Detection – This class includes spatial- and frequency-domain target detection in scenes collected from a wide range of sensor types. Applications include automated exploitation and cueing systems.

Database Management – This class includes algorithms for index maintenance, storage management, and content-based query processing. Applications include sensor data archive management and geographic information systems such as the Dynamic Database for Battlefield Situation Awareness.

From these classes, five algorithms were selected–two from Model-Based Image Generation, two from Target Detection, and one from Database Management. Analyses of these algorithms which suggest their possible performance gains are included below.

2.2 ALGORITHM ANALYSES

Each of the selected applications was analyzed, with a specific interest in the identification of computational bottlenecks in the algorithms and potential performance improvements offered by DIS architectures. Fragments of the algorithms suitable for benchmark implementation were identified, and from these the specifications found in Section 4 of this document. The remainder of this section presents individual analyses of the five selected algorithms.

2.2.1 Method of Moments

The first class of algorithms chosen for inclusion in the DIS benchmark suite are Method of Moments (MoM) algorithms, which are frequency domain techniques for computing the electromagnetic scattering from complex objects. MoM algorithms require the solution of large dense linear systems of equations. Traditionally, MoM algorithms have employed direct linear equation solvers for these systems. The high computational complexity of the direct solver approach has limited MoM algorithms to low frequency problems. Recently, fast solvers have been introduced which have low computational complexity. The potential of these fast solvers to enable MoM algorithms to solve larger problems at higher frequencies is ultimately limited by the speed of main memory. Thus, fast MoM algorithms may benefit from the Data-Intensive Systems research effort.

In MoM algorithms the integral equation form of the Helmholtz equation is discretized by expanding the surface currents induced by the applied excitation in N basis functions. Then N test functions are used to convert the integral equation to a dense linear $N \times N$ system that takes the form $Z \bullet J = V$. Generally, N increases as the square of the frequency, and for typical problems, N is greater than 10,000. In traditional MoM algorithms, which first appeared in the late1960's, the dense linear system $Z \bullet J = V$ is solved by a direct linear equation solution algorithm, which may be composed as an in-core or out-of-core solver. On modern parallel computers, the direct solvers may be extended to work on shared or distributed-memory architectures.

The advantage of MoM algorithms is that they are exact representations of Maxwell's equations and highly accurate simulations are possible. The disadvantage of the traditional MoM algorithms is that the methods are computationally intensive, especially as the frequency goes up. The computational complexity of traditional MoM algorithms includes $O(N^2)$ integral evaluations to compute the matrix Z and $O(N^3)$ arithmetic operations to solve the system $Z \bullet J = V$ for J. The memory requirement for traditional MoM algorithms is $O(N^2)$. For these reasons, the traditional MoM algorithms are generally used only for low frequency problems. Although traditional MoM algorithms have been highly optimized on a variety of high-performance computing machines, the largest problems solved so far are for N on the order of 100,000.

Recently, new fast MoM algorithms based on fast, iterative linear equation solvers have been introduced. The iterative solvers rely on numerically stable and rapidly converging iteration procedures, such as the preconditioned GMRES method [Saad]. Fast matrix-vector multiply algorithms are used to compute products of the form $Z \bullet X$ used in the iterative procedure. The computational complexity of the fast MoM algorithms is $O(N\log N)$. The memory requirement for the fast MoM algorithms is $O(N)$. This is a remarkable reduction from the $O(N^3)$ computational complexity of the traditional MoM algorithms, and potentially, allows the solution of much larger problems at higher frequencies.

Rohklin [Rohklin-1, Rohklin-2] has introduced new fast MoM algorithms for the Helmholtz equation, which use iterative linear equation solvers and the fast multipole method (FMM) for fast matrix-vector multiplies. To compute products of the form $Z \bullet X$, the Z matrix is not formed or stored, rather the product $Z \bullet X$ is viewed as a field and approximately evaluated by the FMM. The mathematical formulation of the FMM is based on the theory multipole expansions and involves translation (change of center) of multipole expansions and spherical harmonic filtering. The computational complexity of these new methods is $O(N\log N)$ and the memory requirement is $O(N)$.

Building on the FMM approach, Dembart, Epton and Yip [Dembart-1 to 4] at Boeing have implemented a fast MoM algorithm in a production grade electromagnetics code used by the company for radar cross-section (RCS) studies. Problems for which the number of unknowns is on the order of 10,000,000 have been solved with this code. Boeing's fast solver uses the preconditioned GMRES iterative method, which requires only the calculation of products of the form $Z \bullet X$, combined with a multilevel FMM for fast matrix-vector multiplies.

The potential of the fast MoM algorithms to solve larger problems at higher frequencies, which results from their low computational complexity, is impacted by two memory bottlenecks encountered by fast solvers: low reuse of data and non-unit stride memory access.

We introduce the issue of low reuse of data by first considering the direct solvers used in the traditional MoM algorithms. Since the computational complexity is $O(N^3)$ and the memory requirement is $O(N^2)$ for direct solvers, the ratio of computation to data access is $O(N)$. For typical problems solved by the traditional MoM algorithms, where N is greater than 10,000, data reuse is high. When data reuse is high, cache is an effective tool for enhancing processor performance. The direct solver, in-core or out-of-core, can be organized so that a block of data is placed in cache and then reused from cache. This effective use of cache makes the computer perform as if all the memory is as fast as the cache memory. Similarly, direct solvers can be optimized for shared- or distributed-memory architectures.

For the fast MoM algorithms, where the computational complexity is $O(N\log N)$ and the memory requirement is $O(N)$, the ratio of computation to data access is $O(\log N)$. Indeed, implementation of Rohklin's translation theorems shows that for the translation operations, which are key to the FMM, the ratio of memory access to computation is 3-to-1. Thus, cache cannot be used to enhance processor performance, and the speed of fast MoM algorithms is ultimately limited by the speed of main memory.

In addition to the bottleneck resulting from the low reuse of data, fast solvers based on the FMM face a second memory related bottleneck. The FMM relies on the numerical implementation of spherical harmonic filtering. The filter operates on rectangular arrays of data in three stages. The arrays are accessed first by rows, then by columns, and finally, by rows again. In the second stage, it is necessary to access memory locations that are not consecutive. Thus, the speed of the fast MoM algorithms based on the FMM is ultimately limited by the speed of accessing main memory with non-unit stride.

Fast MoM algorithms, based on efficient iterative linear equation solvers, have the potential to compute the electromagnetic scattering from complex objects at frequencies 10 to 100 times higher than possible with traditional MoM algorithms. As pointed out above, the ultimate potential of these fast MoM algorithms is limited by two memory-related bottlenecks: low reuse of data and non-unit stride. For these reasons we have chosen to use Boeing's fast solver, based the preconditioned GMRES iteration method and the FMM for fast matrix-vector multiplies, as the basis for the *Method of Moments Benchmark*. The key FMM kernels represented in the benchmark are the translation operations and spherical harmonic filtering.

2.2.2 Simulated SAR Ray Tracing

The simulation of Synthetic Aperture Radar (SAR) provides a cost-effective alternative to real data collections. In contrast to deployed sensors systems, whose operational parameters are fixed, computer simulations allow continuous variation of system and scene parameters. They have been used to simulate the performance of hypothetical sensors systems and to predict the signature of targets from a large number viewing angles as well as target signature that are inaccessible. These simulated target signatures have been used to design, test, and have been included as part of ATR systems.

Phenomenological models of targets and backgrounds and their interactions are the theoretical foundation of the computer simulations. For example, both image-domain and phase-history-domain approaches have been used to simulate synthetic aperture radar (SAR). The image domain approach uses a generalization of the physical optics approximation to compute target scattering. Such an approach is very amenable to use with a solid geometry target model sampled by ray casting. The phase history domain approach uses a variety of methods to compute target scattering: physical optics (PO), physical theory of diffraction (PTD), method of moments (MoM), and others. Hybrid implementations of these two methods have also been developed. The SAR simulation method analyzed for this benchmark is based on the image domain approach.

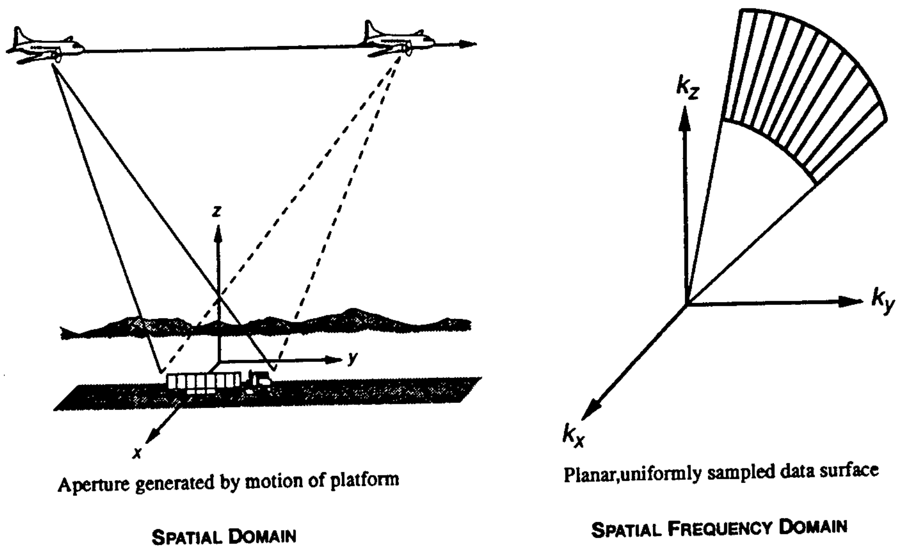


Figure 2-1: A typical geometry for airborne collection of SAR data relative to a specific ground site

A typical SAR collection geometry is illustrated in Figure 2-1. Assume far field conditions and a narrow-band signal. Let α and β denote, respectively, the receive and transmit polarizations of the radar, *u(t)* the transmitted waveform, and γ(r') the so-called SAR reflectivity of the scene. The radar return signal is generally represented as

$$v_{\alpha\beta}(t) = K \int_S \gamma_{\alpha\beta}(r')u\left(t - 2\tfrac{R}{c}\right)ds'$$

where S is the illuminated portion of the scene, R = |r - r'| is the distance from the radar to the point r' on S, c is the speed of light and K is a system constant. In essence, the model is based on the argument that the return from a differential surface element *ds'*, located at r', is a replica of the transmitted signal. This signal is delayed in time by the two-way propagation time from the radar to r' and back, and modified by the reflectivity of the surface element. It can be shown that such a model is consistent with physical optics, and an explicit formula for γ can be obtained.

It is customary to demodulate ν(*t*) by mixing with a reference signal *h*(*t*), yielding

$$s(t) = h(t)v(t)$$

Let Γ(*f*) denote the spatial Fourier transform of γ(*r*):

$$\Gamma(f) = \Im\{\gamma(r)\}$$

A single range record $s(t)$ can be interpreted as corresponding to the values of $\Gamma(f)$ over a radial line segment in the spatial frequency domain. The complete record of $s(t)$ for a sequence of pulses transmitted and received at positions along the platform trajectory constitute a so-called phase history.

The fact that the collected data corresponds to the Fourier transform of the reflectivity density suggests that a reconstruction (image) of the reflectivity can be obtained by inverse Fourier transformation of the data. It will be at best a partial reconstruction because we have only partial data. For a linear trajectory, the phase history represents a planar surface in the spatial frequency domain over which the value of $\Gamma(f)$ has been sampled. The most that can be obtained is a two-dimensional image. Letting $A(f)$ denote a weighted, two-dimensional processing aperture over the data surface, the SAR image formation process is given by

$$g(r) = \Im^{-1} \{ A(f)\,\Gamma(f) \}$$

Additional insight is gained by noting that the image formation process is mathematically equivalent to the convolution

$$g(r) = a(r) * \gamma(r)$$

where

$$a(r) = \Im^{-1} \{ A(f) \}$$

is known as the spatial impulse response.

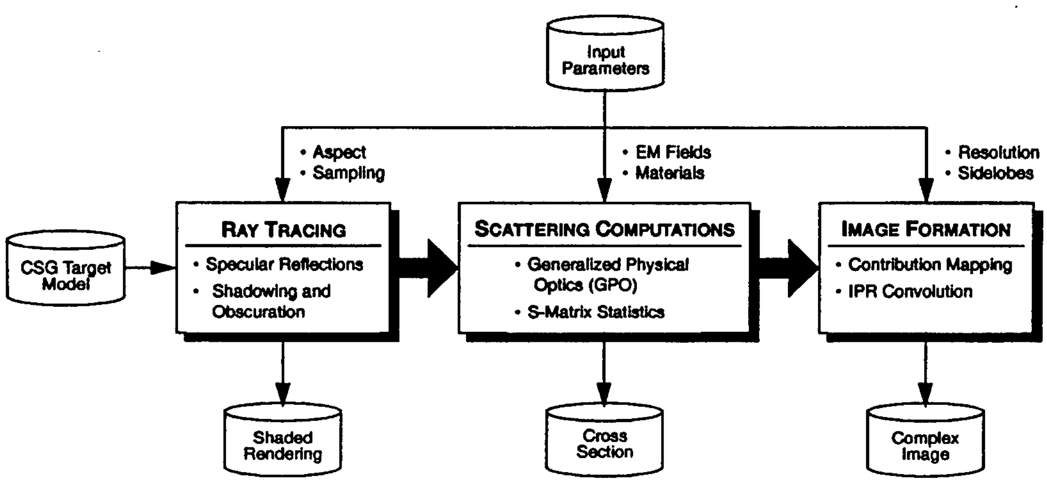


Figure 2-2: Block diagram of the generalized physical optics SAR simulation

Simulation of the SAR system can be achieved by synthesis of the phase history followed by aperture weighting and inverse transformation, or by direct computation of the reflectivity density followed by convolution with the spatial impulse response. Both methods have been implemented in SAR simulation programs like X-PATCH and RADSIM.

In the process of analyzing this approach, the simulated SAR technique can be broken down into three steps as seen in Figure 2-2.

First, is the process of sampling a scene database made of polygons, splines, and Constructive Solid Geometry. A ray tracing system is used to accomplish this sampling, by simulating how the radar energy bounces through the scene.

Ray tracing is a process where rays are fired from a viewing window into the scene and recursively traced through their specular reflections. Each ray is defined as vector with a starting position at the sensor and a direction defined by the pixel it passes through on the viewing window, in this case our synthetic aperture. Each ray is tested against all objects in the scene to see if an intersection can be found. The process of finding the intersection involves finding the roots of a system based on the combination of the vector and object. If multiple intersections are found the closest intersection is used. Once an intersection is found, the object ID and the intersection coordinates are recorded. In addition, several other properties at the intersection point are determined. These are surface normal, curvature, surface material type, and length of the ray from the intersection point to the origin of the ray. Using the surface intersection normal and the incoming ray, a reflected ray is calculated along the perfect specular reflection direction. This ray is fired from the current intersection point and the next intersection is found. This recursive process continues until either the ray leaves the scene, or a preset number of reflections are found. The intersection results of each original ray and all of its reflections create a ray history that contains all the intersection information, normally stored in a linked list. The output of the ray-tracing section is a ray history for each pixel in the image plane.

In programs like X-PATCH, the ray-tracing portion of the process consumes 50% to 60% of the total computation time. With this being the major time component in the SAR simulation process, it is a prime candidate for parallelization. Parallel ray tracing has been investigated by several researchers and is not a simple problem. This process will be the major thrust of the benchmark effort for simulated SAR imagery.

The second step is the process of converting the ray-traced information, the ray history, into the electromagnetic (EM) response of the sampled scene data. Here each ray path is analyzed to generate a fully polarimetric EM response solution. This is a linear process and does not consume a large amount of time. This step, in the SAR simulation, would be a trivial process to parallelize because each ray history is independent of all the others. Due to the small amount of time and the simplicity of parallelization, this portion of the process is not considered as part of the benchmark.

The final step in the simulated SAR process is converting the 2-D array of EM responses into complex images. This is accomplished by mapping the 2-D array of EM responses into the slant plane. This slant plane image is then convolved with a system Impulse Response (IPR) to form a complex image that can be detected and viewed.

This final step is a unique combination of processes, from the viewpoint of parallelization, and does present the second-highest consumer of CPU time. Creating a parallel version of this section of the process will stress data-passing, as EM responses are mapped onto a rectangular grid called the *slant plane*. This output then runs through a standard convolution. Each of these steps will require different lay-outs of memory and should present some unique problems as a parallel implementation. For this reason, and because this step is a large time consumer, it is part of the simulated SAR benchmark.

2.2.3 Image Understanding

Image processing algorithms represent the third type of algorithm chosen for study. The applications of interest include target detection and classification. A sampling of these algorithms was chosen for this benchmark identifying bottlenecks that are common to image processing applications. The sampling contains algorithms that perform spatial filtering and data reduction. The algorithms selected for the benchmark are a morphological filter component, a region of interest (ROI) selection component, and a feature extraction component. These form the Image Understanding Sequence as shown in Figure 2-3. The morphological filter component provides a spatial filter to remove background clutter in the image. Next, the ROI selection component applies a threshold to determine target pixels, groups these pixels into ROIs, and selects a subset of ROIs depending on specific selection logic. Finally, the feature extraction component computes features for these selected ROIs.

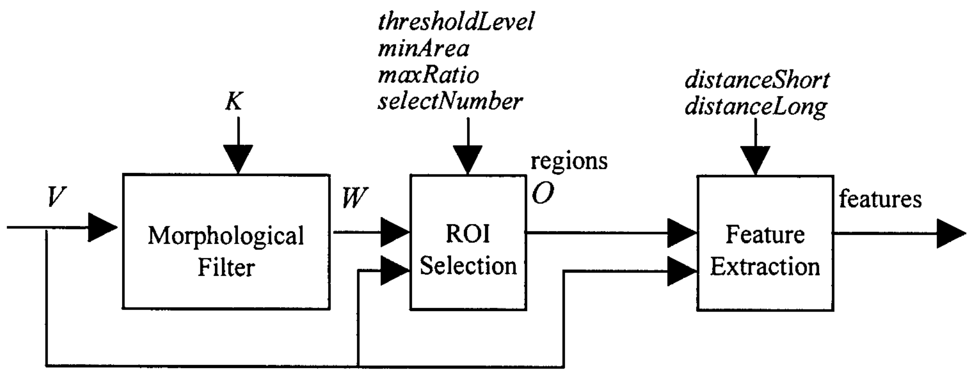


Figure 2-3: Image Understanding Sequence

Transformations that generate images from symbolic input, as well as Fourier Transforms, were excluded, since these are addressed in other portions of the Benchmark Suite.

The input required by the sequence is a set of parameters and an image, *V*. The first step in the sequence is a spatial morphological filter component generating image *W*. Then, the ROI selection component applies a simple threshold and groups connected pixels into ROIs (or targets) contained in image *W*. This component then computes initial features for each ROI in image *W*, and selects candidate ROIs, depending on the values of these features. These selected ROIs are stored in object image, *O*. The initial features for each selected ROI are stored in the list *regions*. Lastly, the feature extraction component computes additional features for the selected ROIs. The final output is a feature list, *features*, containing all the features calculated for each selected ROI. Details regarding the sequence can be found in Section 4.2.3.

Each algorithm has two associated costs: *operational* and *pixel addressing*. The operational cost is a measure of the computational burden placed upon the processors to execute the algorithm, and pixel addressing cost is a measure of the amount of memory usage or access that is required. A brief description and analysis of each component, including its bottlenecks, follows.

The morphological filter component chosen for the benchmark is a relatively straightforward procedure, designed to remove background clutter and retain objects of interest. The total cost of the morphological filter is determined by assuming the kernel is applied over the entire input image, although in practice the kernel is usually only applied over a subset of the image (the input image less a portion at the edges). The address-to-operation ratio is approximately the same for each approach. The filter utilized in this benchmark includes three distinct phases: erosion, dilation, and difference. The number of operations for the filter is

$$\mathrm{size}(V)[4\,\mathrm{size}(K)+1]$$

where V is the input image, K is the kernel, and size(X) is the total number of pixels in X. The operational cost consists of two multiplies, one subtraction, one minimum comparison, and one maximum comparison. The number of pixel addresses is

$$\mathrm{size}(V)[4\,\mathrm{size}(K)+5]$$

where the kernel and input image are accessed multiple times as the kernel is applied over the input image. The address to operation ratio is then

$$(4\,\mathrm{size}(K)+5)/(4\,\mathrm{size}(K)+1)$$

which is bounded in the range {1, 1.8}.

The ROI selection component of the sequence involves a threshold phase, a connected-component phase (where detected pixels are grouped into objects), an initial feature extraction phase, and a selection phase (where ROIs are selected based on the values of the initial features). The initial feature extraction phase measures five characteristics of the object region. Three of these–*centroid, area*, and *perimeter* –are descriptive of the shape and location of the ROI. The other two–*mean* and *variance* –are statistical measures of amplitude over the pixel population of the ROI. The threshold phase has an address-to-operation ratio of two. The operational and pixel addressing costs associated to the connected component phase, the initial feature extraction phase, and the selection phase vary greatly, depending on the implementation and the data involved, so no analysis of these costs is provided here.

After selecting ROIs, additional features are calculated. These give a rough measure of the texture of each ROI. As discussed in [Parker 97], a gray-level co-occurrence matrix (GLCM) contains information about the spatial relationships between pixels, by representing the joint probability that a pixel with a given value will have a neighboring pixel at a particular distance and direction with another chosen value. Since this matrix is square, with dimensions equal to the number of possible pixel values, it provides more information than can easily be analyzed. Statistical descriptors of the co-occurrence matrix have been used as a practical method for utilizing these spatial relationships. Furthermore, [Unser] designed a method of estimating these descriptors without calculating the GLCM, instead using sum and difference histograms. The descriptors chosen as features for this benchmark are *GLCM entropy* and *GLCM energy,* and are defined in terms of a sum histogram, *sumHist,* and a difference histogram, *diffHist.* These descriptors are calculated for each of two distances and four directions.

It is typical in target detection systems to calculate many features to be used in a target recognition step. The ideal is to choose the fewest and cheapest features possible that provide the best detection result. The cost for the feature extraction component is dependent upon the number of features or targets present in the input image which can range from zero to several thousand in typical applications. This makes the algorithm very difficult to execute efficiently, since many features will have a high computational cost with a small memory access cost, while a few will have a low computational cost with a high memory access cost. Thus, an *a priori*-implementation for feature extraction is generally not possible. Consequently, there is no analysis provided here of the cost involved to calculate these features.

The two main bottlenecks which occur in typical target recognition applications are the result of manipulations of large amounts of data while expending little computational effort, and of smaller amounts of data in computationally intensive functions. The intent of this benchmark is to represent these bottlenecks within the sequence, so that attempts to remove these bottlenecks may be examined.

2.2.4 Multidimensional Fourier Transform

The Fourier Transform has wide application in a diverse set of technical fields. It is utilized in image processing and synthesis, convolution and deconvolution, and digital signal filtering, to name a few. In fact, the transform is utilized within both the Ray-Tracing and Method of Moments benchmarks described elsewhere in this document. However, special interest in the Fourier transform merits its independent inclusion in this benchmark suite. Specifically, the interest is in the nature of the memory access patterns, which are indicative of a large class of problems.

The multi-dimensional Discrete Fourier Transform (DFT) is defined as

$$F(n_1, n_2, \ldots, n_N) = \sum_{k_N=0}^{N_N} \cdots \sum_{k_1=0}^{N_1} e^{2\pi i k_N n_N / N_N} \cdots e^{2\pi i k_1 n_1 / N_1} f(k_1, k_2, \ldots, k_N) \qquad (2.2.4.1)$$

where f is an input complex multi-dimensional array of size $N = N_1 \times N_2 \times \cdots \times N_N$, and F is the output forward transform of f. The Fourier Transform is rarely implemented directly as Equation 2.2.4.1, since the process would require $O(N^2)$ operations. Instead, the transform can be accomplished in $O(N \log_2 N)$ operations, or less, using one of a series of methods generically called Fast Fourier Transforms (FFT). These FFT methods exploit one or more mathematical properties of Equation 2.2.4.1 to reduce the required number of operations.

The bottleneck associated with DFTs that is of interest here is the non-unit-stride memory access associated with the transform. Part of the subscripts of Equations 2.2.4.1 can be "pulled out" of the summations (i.e., the exponential with the subscript k_N can be pulled outside of the sum over k_{N-1} etc.), which shows that the multi-dimensional DFT can be represented by a series of one-dimensional DFTs:

$$F(n_1, n_2, \ldots, n_N) = F_N\left(F_{N-1}\left(F_{N-2}\left(\cdots F_1\left(f(k_1, k_2, \ldots, k_N)\right)\right)\right)\right) \qquad (2.2.4.2)$$

where F_k is a one-dimensional DFT over the specified index. The aspect of Equation 2.2.4.2 to note is that for whatever memory access the inner loops attempt, the outer loop will always be "opposite" or irregular, which prevents a unit-stride access. Rearrangement of the summations or manipulation of the equations can alleviate this memory access bottleneck to some extent, but some non-unit-stride access is present with most DFT implementations.

In order to simplify the implementation and specification of this benchmark, the DFT is limited to three-dimensional transforms only. The implementation of a 3-D transform is complex enough to give an indication of the performance of the architecture on higher dimensional transforms, but simple enough to be relatively easy to implement. The inclusion of one- or two-dimensional transforms would not significantly add any other performance information regarding the candidate architectures. In addition, one- and two-dimensional input can be approximately tested by specifying the length of the remaining dimensions of the array to be one.

2.2.5 Data Management

The fifth area in which the DIS benchmark suite attempts to measure performance improvement is in data management, specifically in the area of DBMS. Applications for traditional DBMS have been dominated by archival storage and retrieval of large volumes of essentially static data. Some newer applications, such as the Dynamic Database for Battlefield Situation Awareness, demand management of complex, dynamic indices in addition to the data.

The objective of this benchmark is to measure the performance improvement of a given hardware configuration for certain elements of traditional DBMS processing. Performance improvements due to sophisticated database design or special software implementation are avoided and not intended to be part of the benchmark. This benchmark focuses on two weaknesses of conventional DBMS implementations: index algorithms and ad hoc query processing.

Large volumes of data in a DBMS are typically referenced by an index structure. The index can be used instead of brute-force searches over all the data when a query is made. The index defines one or more elements of the data entries as key values. Thus, the key values are specified in advance, and the DBMS maintains a separate index structure based on them. The index is used to accelerate query processing by minimizing the amount of data that must be accessed to satisfy the query.

Two assumptions typical of conventional algorithms are that the data will be predominately static, and that operation can be suspended for index maintenance. Neither assumption holds for the Dynamic Database or other dynamic information systems, and current applications drive standard indexing schemes into frequent wholesale index regeneration, yielding unacceptable performance.

The index structure allows efficient searches over a database when the query can use a pre-defined key value. Queries which do not use a key value are called ad hoc, non-key, or content-based queries. This query type requires a brute-force search over all database entries. Conventional applications usually process an ad hoc query in two stages: an index-based search is used for the index keys in the query formulation, if any, and brute-force search is performed on the results of the index-based search. These brute-force searches are a bottleneck in a typical DBMS. The performance impact of non-key queries can be reduced by parallel searches of the data, which may be applicable to specific hardware architectures, or by partitioning the data.

Partitioning schemes provide an additional performance boost for a general database design where the primary objective is to separate areas of the database into logical sections, each of which is then indexed by its own scheme. The partition allows more efficient searches, when the sections have been chosen well, or when an optimal scheme is known *a priori*. It also supports parallel searches across partitions.

Bottlenecks traditionally associated with DBMS primarily occur in query processing, and the majority of work done to enhance performance has been in this area. Much of this query optimization has increased the query response speed at the expense of maintaining the index over the lifetime of the database. By definition, an index requires an increase in overhead or up-front processing in favor of quicker, cheaper searches. Typical command operations such as *insert* and *delete* have generally not been optimized. This reiterates the implied assumption of the existence of periods during operation when user interaction can be suspended to deal with index management. The cost associated with index management over the operation life of the database represents a new measure of performance for advanced data management applications, and a corresponding new bottleneck.

The indexing method chosen for use within this benchmark is an R-Tree structure. The R-Tree index allows the key to represent spatio-temporal data, which makes the R-Tree particularly applicable to geographic information; it is commonly supported by database vendors. The R-Tree structure is as close to a de-facto standard for representing such data in a database context as exists today.

The R-Tree index is a height-balanced tree containment structure, that is, nodes of the tree contain lower nodes and leaves. Thus, the tree is hierarchically organized and every level in the tree provides more detail than the previous level. The indexed data object is stored only once, but because of the containment structure, keys at all levels are allowed to overlap. This may cause multiple branches of the R-Tree to be searched for a query whose search index intersects multiple nodes.

A general measure of index maintenance cost for separate command operations is the number of node accesses required for each operation. Other measurements of cost become increasingly software-dependent, and are avoided in this analysis. A generic R-Tree implementation, which is given later in this document, has three command operations to measure: insert, delete, and query. Because the R-Tree is a height-balanced structure, the total number of paths for a full tree is given by:

$$N = \sum_{k=1}^{h} 2^{k-1} F$$

where N is the number of paths, h is the height of the tree, and F is the *fan* or *order* of the tree. Traditional performance measures have focused on the query response: for the generic R-Tree the minimum number of node accesses is h, which is expected from a height-balanced tree, and the maximum number of node accesses is N, or a complete node search over all possible paths. The maximum number is unique to the R-Tree or similar overlapping index trees and represents a significant bottleneck. The problem is exacerbated for improperly managed index structures, and can be alleviated by efficient software implementations and improved hardware architectures which allow more efficient or parallel searches.

Index management over the operation of the database represents a new type of bottleneck for advanced applications. The cost of maintaining the index can be estimated in the same manner as for query commands, by determining the number of node accesses required to complete the command in both the best and worst cases. A descriptive estimate of the average case is also given, with the caveat that the average case is highly implementation-dependent, and will vary for each system.

The insert operation has three separate phases: a search over all paths, insertion (which may cause node splitting), and index key adjustment. The best case occurs when insertion does not require node splitting and no parent keys need to be adjusted; this yields a cost of N node accesses. The worst case does require splitting along each parent, and all parent keys are adjusted; this yields a cost of $N+2h$ node accesses. An average insert would tend to require parent key adjustment and periodic node splitting. Thus, the average insert cost would tend towards the maximum cost.

The delete operation has two phases: a search for the data to be deleted, and a possible key readjustment. The best case has a cost of h node accesses, which represents no key adjustments and an immediate "one" path search for the data. The worst case has a cost of $N+h$, which represents a full search of the data and an all parent key adjustment. The average cost of a delete operation tends to the minimum case, since the operation would include key adjustment but probably not a full search.

The costs of the insert and delete operations are greater than or equal to the query operation in both the best and worst cases. Thus, index management over the operational life of the database represents a significant performance bottleneck when the data is dynamic.

This benchmark has been developed to measure the performance improvements of new hardware architectures for both index maintenance and non-key queries, which represent the two significant performance bottlenecks. One goal is to remove or "level" the algorithmic component over all of the architectures, without preventing any new or unique software implementations that would allow a significant performance improvement due to exploitation of special hardware features. This is done by defining the benchmark as the implementation of a highly simplified database with a specific index structure. The database supports only three simple aggregate data objects whose primary difference is in size. The use of different sizes of data objects is intended to prevent optimization of the implementation for an object of a specific size, and the sizes themselves were chosen to prevent similar multiples. The objects are aggregate in that they contain a set of data attributes or parts which are linked together as a list. An ad hoc query uses an attribute of the object for non-key searches. This type of search with simplified objects is relatively simple to implement, but is representative of more complicated database behavior such as object traversal. This benchmark requires the use of the R-Tree structure, but the participant is encouraged to modify or develop additional implementations tailored for new architectures.

The DIS benchmark metrics provide a measurement of the candidate architecture's ability to handle the "highest" load when the number of users is large and the system resources are taxed to their limits. The benchmark simulates this maximum resource utilization by issuing the index commands in a batch rather than a stream mode. A stream mode would more closely mimic a "real" DIS application, allowing for multiple users and possible "down" time for index maintenance. However, this benchmark is primarily interested in the extreme condition, where down-time, in which a database can perform index maintenance with no cost to the users, is assumed not to exist. The performance on successful completion of the entire data set with its multiple commands is the primary metric of this benchmark, and this must include the time required for index maintenance since this will directly affect the users under extreme conditions. Participants are allowed to introduce artificial lags to the command input to simulate a stream mode, but the times reported for individual command completion and overall set completion must include the added lag times.

3. PROCEDURES

This section provides information on the procedures to be used when employing these benchmarks. The primary purpose of the section is to answer the question, "How does one use this benchmark?" It begins with an overview, then describes the procedures which are common across all four benchmarks in this set. Metrics, measurement, reporting, data types, and precision are also addressed.

3.1 OVERVIEW

Procedures are a critical element of benchmarking. To be useful, benchmarks must be approached uniformly and analyzed consistently. According to [Honeywell], the following questions must be addressed by benchmarking procedures:

How should baseline performance metrics be established, against which to compare the other results?

How should the operations in the benchmark specification be performed?

How can it be ensured that an implementation is solving the intended problem?

How should measurements be made?

How should benchmark results be reported so that anyone examining the results has sufficient information to interpret them correctly?

3.2 BENCHMARKING PROCEDURE

The following procedure should be executed by any group or individual wishing to generate an implementation of this benchmark set. The sequence was published in [Honeywell], and applies to all five of the benchmarks in the set. It is presented here modified only to the degree necessary for relevance to this set.

Step	Action
1	Review the background and all procedures as specified in this document. These capture the common aspects of all benchmarks.
2	Review the benchmark specifications, as given in Section 4 of this document, noting any restrictions associated with the benchmark development activity. Understand the metrics of interest, the acceptance test, and the information that needs to be reported.
3	Develop a benchmark implementation in accordance with the benchmark specification. This step can take one of two forms: simple compilation, with no source code modification, in the case of an 'un-optimized' code test, or manual optimization, to the degree desired by the participant. In this case, the specific steps or operations to be performed are particular to the benchmark being implemented. A precise description of the steps is provided in each respective benchmark specification. Any restrictions regarding the steps are also provided. For both cases, baseline source code, written in the C programming language is

	provided.
4	Tabulate any information required by the Metrics portion (Section 3.3) of this document.
5	The process used to measure benchmark metrics is important in the interpretation and reproducibility of results. Section 3.3.4 provides a description of the allowable techniques for measurement. Timing must be performed as specified in this document. Benchmark runs should be repeated a sufficient number of times as to ensure reproducibility of results.
6	The use of acceptance tests is critical to determining whether a specific implementation is deemed valid. For each benchmark run, examine the results and verify that they pass the appropriate acceptance test as defined in the benchmark specifications (Section 4).
7	Reporting of results is a weakness of many existing benchmark approaches. Section 3.5 addresses the topic of results reporting in detail, providing guidelines regarding what information needs to be provided to ensure that adequate interpretation of the results is possible.

Participants are required to measure performance of all five benchmarks using 'un-optimized' code. That is, the baseline code provided for each benchmark must be compiled without any modification, and run 'as-is' to establish performance of the architecture utilizing only automatic optimizations. In addition, participants are encouraged to modify or replace this baseline source code and run the tests again, establishing performance after manual optimizations. This should be done for one of the benchmarks at the very minimum. This process may be repeated as desired, but users are reminded that each level of optimization must be documented in accordance with Section 3.5.1.

Therefore, the above procedure must be followed at least six times–once for each 'un-optimized' benchmark, and once for one 'manually optimized' benchmark. There is no limit to how many times it may be followed. However, the ability of the reviewers to effectively analyze the results must be considered.

3.3 METRICS

Of primary interest for all the benchmarks in this set is the trade-off between 'performance' and 'cost', where performance is focused mainly on maximizing throughput, and cost is focused mainly on programmer labor costs. Of course, there are many other important considerations relating to performance and cost; some important contributing factors are listed here:

Performance	Cost
Maximize throughput (primary)	Minimize programmer labor (primary)
Maximize scalability	Minimize development labor
Minimize power consumption	Maximize use of OTS parts
Maximize robustness	Minimize part count
Maximize implementation flexibility	Maximize ability to retrofit
	Minimize volume and weight

To quantify these factors, metrics are specified for each benchmark. These are only concerned with the performance aspect of the trade-off. It is expected that the cost aspect will be addressed in the *Architectural Description* and *Comments* portions of the reports. While each benchmark will require different measurements of performance, all metrics are intended to quantify throughput. How these measurements vary over the range of input data sets gives some notion of scalability for a specific configuration. How the configuration itself can be scaled is another issue to be addressed in the *Architectural Description*.

Implicitly, the implementations are expected to provide correct output to even be considered. This requirement is an element of each acceptance test, and is therefore not a metric in need of evaluation.

The energy spent by implementers laboring in the development of each benchmark implementation is of special interest. As this is ultimately difficult to measure accurately, reviewers will rely on participant's candid reporting on this subject. A frank summary of the required skills, labor expended, and problems encountered during the process would be of great benefit to those establishing the utility of a given design.

Although peripheral devices vary greatly in access speeds and communication throughput rates, it is desirable to understand the limitations on throughput induced by the architecture. Therefore, for all benchmarks, the time spent reading from input and writing to output should be included in time-for-completion metrics, but recorded separately where possible. Participants should comment on I/O features or limitations in their *Architectural Descriptions*.

Power consumption is also an important metric. Again, the early stages of development under the DIS program might make accurate quantification of power consumption impossible. However, participants are expected to include their best estimates of the power required for each benchmark in the suite. Measurement methods employed should be detailed in the report, as it is anticipated that no specific methodology may reasonably be imposed.

Although scalability with respect to problem size is largely addressed by the spectrum of input sets provided with the benchmark, scalability with respect to processor or memory configuration can only realistically be addressed by qualitative analysis. Participants should address this issue for each benchmark in their reports.

Additional metrics for each of the benchmarks are described in the following subsections. This information is also included in the benchmark specifications (Section 4). These, and the above basic metrics should be considered the minimum required for a report. It is in the participant's interest, however, to supply a complete report, with all the details relevant to evaluators concerned with DIS applications.

3.3.1 MoM Benchmark Metrics

The metrics for the *Method of Moments Benchmark* consist of three measures: performance, scalability with respect to problem size, and scalability with respect to processors. The metrics are described in detail in Section 4.2.1.5.

The most important of the three metrics is performance, which is measured in wall-clock time. The primary measure of performance is the total wall-clock time for the calculation of the far-field by the multilevel FMM. The secondary measure of performance is the breakdown of the total time into the total time for all translation operations and the total time for all spherical harmonic filtering/synthesis calculations. The tertiary measures of performance are the breakdown of the secondary measures into the total time for each of the six steps in the multilevel FMM as specified in Section 4.2.1.2.2.

3.3.2 Simulated SAR Ray Tracing Benchmark Metrics

The primary metric for the simulated SAR ray-tracing benchmark suite will be total execution time. Secondary metrics will be scalability (how does adding more processors effect the timings and how do larger data sets effect the timings) and load distribution (how is the workload distributed among the processor). These secondary metrics are important measures for the ray tracing part of the benchmark. The major problem with parallel implementations of ray tracing is in achieving a constant work load and maintaining scalability.

3.3.3 Image Understanding Benchmark Metrics

The primary metric associated with the image understanding benchmark is total time for accurate completion of a given input data set. A series of secondary metrics for the individual times of the processing operations is also required. See Section 4.2.3.5 for more detail.

3.3.4 Multidimensional Fourier Transform Metrics

There are three metrics for this benchmark. The first, and primary, is the total time required to complete the input set. This should include the time for each transform test as well as the I/O time required to load the randomly generated input, and output the result. The total time should not include the time necessary for the generation of the random data. The second metric is the time required to complete the individual transform tests. Again, this time should include any I/O time for loading of data and output of results. The third metric measures the "mflops" [Johnson] of the individual transform tests. The "mflops" for a given transform is defined to be

$$"mflops" = \frac{5(X \times Y \times Z)\log_2(X \times Y \times Z)}{(\text{time for one DFT in } \mu\text{s})}$$

where X, Y, and Z are the lengths of the first, second, and third dimensions, respectively. The rational behind using this metric is to provide a reasonable comparison between different architectures, implementations, and transform sizes. Note that "mflops" are not equivalent to MFLOPS (millions of floating-point operations per second), but are instead an estimate of that value that assumes a common baseline number of operations for any implementation as

$$5(X \times Y \times Z)\log_2(X \times Y \times Z) + \vartheta(N)$$

which is the radix-2 Cooley-Tukey FFT[Cooley]. This third metric is common in the FFT literature and for more discussion of the reasoning behind the metric, the reader is referred to [Johnson].

3.3.5 Data Management Benchmark Metrics

The primary metric associated with the Data Management benchmark is total time for accurate completion of a given input data set. A series of secondary metrics are the individual times of the command operations: *Insert*, *Delete*, and *Query*. Best, worst, average, and standard deviation times should be reported for all operations for each data set.

The time for a non-response command operation to complete is defined as the time difference between the time immediately before the command is placed in the database input queue and the time immediately before the next command is placed in the same input queue. This time difference is essentially the rate at which each line of the input data set is read and executed. This definition is applied to the *Insert* and *Delete* command operations. The time for a *Query* command operation to complete is defined as the time difference between the time immediately before the command is placed in the input queue to the time immediately after the response is placed in the output queue.

3.4 MEASUREMENT PROCEDURES

When measuring performance of a benchmark implementation, the following considerations must be made:

Actual platform measurements are preferred over simulated results. It is understood that early iterations through the benchmarking process will necessarily be based on simulation, but these must give way to measurements of actual systems for reliable determinations to be achieved.

If simulations are used, a description of the model and tools used, and the bases for the timing values, should be provided.

All data sets should be used. They have been provided in a range of sizes, so as to test fixed-system scaling effects resulting from limited-resource optimizations. Should particular data sets be unusable for some reason (e.g., the dataset requires more memory than that which is available), the reason should be reported.

There may be no recompilation or manipulation of the software or hardware between runs producing final measurements. Recall that quantifying the effects of system design decisions is one of the goals of this effort. Therefore, the environment must be consistent throughout the tests to ensure validity of measurements relative to one another.

Tests should be repeated enough times to ensure reproducibility.

As the DIS effort is primarily concerned with memory issues, measurement of time to perform I/O operations shall ideally be factored out. However, because the relative need for–and speed of–I/O is determined by the architecture, these times should be measured and included in the report. If possible, the time for these operations should be noted, so they can be excluded when appropriate.

3.5 SUBMISSION OF RESULTS

3.5.1 Required Elements

Participants are expected to supply the following items as a result of their tests:

Item	Description
Architecture description	A detailed description of the hardware and software environments utilized during testing should be supplied. The description should be sufficient that strengths and weaknesses of the architecture pertinent to the benchmarks can be understood. Known performance measures such as bisection bandwidth and feature size should be included. Limits of the architecture (e.g., maximum of 32 processors, or maximum clock rate of 100Mhz) should be identified, and if predicted performance is to be considered, it must be justified in the *Comments* section of the report. As it is unwise to compare raw timings, even for similar architectures, without considering the differences in technology between the systems, this description is critical to the process, and should be organized, detailed, and complete.
Source code	If modifications are made to the baseline source code in support of optimized performance, the revised source code used during testing should be supplied, along with corresponding documentation of the changes, and detailed documentation of the code compilation, assembly, and execution.
Implementation documentation	A detailed record of the implementation, including rationale and approach to optimizations, is expected. This is particularly important when deviations from the baseline code are employed, or when problems in implementation are encountered. An accurate account of the labor required to implement each benchmark is required.
Output data	Output data sets should be made available. Any deviations from the output data specification should be explained.
Measurements	Performance figures for each applicable benchmark should be supplied, along with a description of how they were obtained. Any missing measurements should be explained. Metrics in addition to those required by this specification are encouraged, but they must be accompanied by documentation of how they were gathered, and how they are pertinent to the analysis. See Section 3.3 for more information.
Comments	Participants are encouraged to include any other information pertinent to the benchmarking process, including explanations of special circumstances, or recommendations for improving the benchmark. To be considered, theoretical performance of an unbuilt architecture should be given and justified. Particular attention should be given to the scalability of the architecture with respect to each of the benchmarks in the suite. Results from implementations of other benchmarks are welcomed, also, though these should be sufficiently delineated so as not to obscure the data directly relevant to this benchmark.

3.6 COMMON DATA TYPES

The reference to several common data types used throughout this document and the accompanying specifications are described in this section. The descriptions given here apply to all data type references unless specifically noted.

<table>
<tr><th>Type</th><th>Description</th></tr>
<tr><td>byte</td><td>A byte consists of eight contiguously stored bits, with bit 0 being the least significant bit (LSB) and bit 7 being the most significant bit (MSB). A signed variant uses bit 7 as a sign bit.</td></tr>
<tr><td>char</td><td>A char represents a 7-bit ASCII character with a decimal range of 0 to 127 as defined by the ANSI specification, stored in a byte (with bit 7 always set to zero). The term whitespace refers collectively to the characters of space (value 32), horizontal (value 9) and vertical (value 11) tabs, line-feed (value 10), and form-feed (value 12).</td></tr>
<tr><td>short integer</td><td>A short integer is a whole number stored contiguously as one 16-bit word (in 2 bytes). Bits 0:15 contain the integer, with bit 0 being the LSB and bit 15 being the MSB. Bit 16 is the sign bit.</td></tr>
<tr><td>integer</td><td>An integer is a whole number stored contiguously as one 32-bit word (in 4 bytes). Bits 0:30 contain the integer, with bit 0 being the LSB and bit 30 being the MSB. Bit 31 is the sign bit.</td></tr>
<tr><td>float</td><td>A float is a single-precision 4-byte real number stored contiguously as one 32-bit word in excess 127 notation, and has a one-bit sign, an 8-bit biased exponent, and a 23-bit fraction. Bits 0:22 contain the 23-bit fraction with bit 0 being the LSB of the fraction and bit 22 being the MSB; bits 23:30 contain the 8-bit biased exponent with bit 23 being the LSB of the biased exponent and bit 30 being the MSB; bit 31 is the sign bit.

The value of a float is given by:

<table>
<tr><th>Value</th><th>Bit Pattern</th></tr>
<tr><td>$(-1)^{\text{sign}} \times 2^{\text{exponent}-127} \times 1.\text{fraction}$</td><td>$0 < \text{exponent} < 255$</td></tr>
<tr><td>$(-1)^{\text{sign}} \times 2^{-126} \times 0.\text{fraction}$</td><td>$\text{exponent} = 0; \text{fraction} \neq 0$</td></tr>
<tr><td>$(-1)^{\text{sign}} \times 0.0$ (signed zero)</td><td>$\text{exponent} = 0; \text{fraction} = 0$</td></tr>
<tr><td>+INF (positive infinity)</td><td>$\text{sign} = 0; \text{exponent} = 255; \text{fraction} = 0$</td></tr>
<tr><td>-INF (negative infinity)</td><td>$\text{sign} = 1; \text{exponent} = 255; \text{fraction} = 0$</td></tr>
<tr><td>NaN (Not - a - Number)</td><td>$\text{exponent} = 255; \text{fraction} \neq 0$</td></tr>
</table>
This description conforms to the IEEE 754 Floating-Point Arithmetic standard and the reader is referred to it for further information including minimum and maximum values.</td></tr>
<tr><td>double</td><td>A double is a double-precision 8-byte real number stored contiguously as two successively addressed 32-bit words in excess 1023 notation, and has a one-bit sign, an 11-bit biased exponent, and a 52-bit fraction. Bits 0:51 contain the 52-bit fraction with bit 0 being the LSB of the fraction and bit 51 being the MSB; bits 52:62 contain the 11-bit biased exponent with bit 52 being the LSB of the biased exponent and bit 62 being the MSB; and the highest-order bit (63) contains the sign.</td></tr>
</table>

The value of the double is given by:

Value	Bit Pattern
$(-1)^{sign} \times 2^{exponent-1023} \times 1.fraction$	$0 < exponent < 2047$
$(-1)^{sign} \times 2^{-1022} \times 0.fraction$	$exponent = 0; fraction \neq 0$
$(-1)^{sign} \times 0.0$ (signed zero)	$exponent = 0; fraction = 0$
+INF (positive infinity)	$sign = 0; exponent = 2047; fraction = 0$
-INF (negative infinity)	$sign = 1; exponent = 2047; fraction = 0$
NaN (Not - a - Number)	$exponent = 2047; fraction \neq 0$

This description conforms to the IEEE 754 Floating-Point Arithmetic standard and the reader is referred to it for further information including minimum and maximum values.

3.7 ARITHMETIC PRECISION

The mathematics in the benchmark algorithms requires the manipulation of values which cannot be stored in finite-precision memory representations. Since the successful completion of a benchmark is determined by a comparison between the output of the benchmark implementation and results provided by the baseline solution, the question of precision and accuracy must be addressed.

The data types used for the input to the benchmarks conform to the IEEE 754 specification, which also specifies the manipulation of these data types to ensure the mathematically expected results and expected properties for finite arithmetic. The output provided with the benchmarks conform to the IEEE 754 standard, as does the baseline implementation. Benchmark participants are not required to implement this specification, but the output of their implementations must have the same level of precision, and perform to at least the same level of accuracy for numeric calculations. This requirement allows the baseline results to be accurately compared over the various architectures and implementations.

4. SPECIFICATIONS

This section provides the specifications of each of the five benchmarks, preceded by an explanation of the approach and intent relative to the specifications. The specifications utilize a common outline, and are intended to contain the information needed by implementers.

4.1 APPROACH

Each specification provided here is intended to be separable from the remainder of the document; it contains all the information needed by a developer charged with building an implementation of a benchmark, with the exceptions of the Common Data Types specification from Section 3.6 and the Arithmetic Precision specification from Section 3.7. Complete specifications of input, algorithm, output, acceptance tests, and required metrics are included.

In several cases, common algorithms used in the solution of the selected problems are optimized for use with traditional systems. For example, certain steps in the Method of Moments algorithm are only present to take advantage of unit-stride memory accesses. While these steps are not strictly part of the solution algorithm, it would be onerous to require participants to independently rediscover them. In some cases, it would require implementers to become experts in the application field. So, the algorithmic descriptions are intended to cover the mathematics of the solutions only. Known optimizations are additionally provided for informational purposes, but implementation of these is not required.

Pseudo-code provided in the algorithmic specifications is intended to provide guidance and clarification of algorithms **only**; it is not intended to represent optimal–*or efficient*– implementations of problem solutions. Similarly, pseudo-code is not intended to represent 'known optimizations' as described above, except when specifically identified as such.

4.2 BENCHMARK SPECIFICATIONS

Detailed descriptions of the benchmark algorithms are included in this section. Any suggestion of the specific implementation of an algorithm is not intentional; all descriptions implying a specific implementation should be viewed as examples only.

4.2.1 Method of Moments

The first class of algorithms chosen for inclusion in the DIS benchmark suite are Method of Moments (MoM) algorithms, which are frequency-domain techniques for computing the electromagnetic scattering from complex objects. MoM algorithms require the solution of large dense linear systems of equations. Traditionally, MoM algorithms have employed direct linear equation solvers for these systems. The high computational complexity of the direct solver approach has limited MoM algorithms to low-frequency problems. Recently, fast solvers have been introduced which have low computational complexity. The potential of these fast solvers to enable MoM algorithms to solve larger problems at higher frequencies is ultimately limited by the speed of main memory. Thus, fast MoM algorithms may benefit from the Data-Intensive Systems research effort.

The scattering of a plane wave of a specified frequency, ω, from an object is given by Maxwell's equations. The Electric Field Integral Equation (EFIE) and the Magnetic Field Integral Equation (MFIE) formulations, which describe the surface current densities induced by an incoming plane wave of frequency ω, are given by

$$\hat{n}\times E(\vec{r})=\frac{1}{i\omega\varepsilon}\hat{n}\times\int_{\vec{r}'\varepsilon S}\left(-\omega^{2}\mu\varepsilon J(\vec{r}')\Psi+(\nabla'\cdot J(\vec{r}'))\nabla'\Psi\right)d\vec{r}' \tag{4.2.1.1}$$

$$2\hat{n}\times H(\vec{r})=J(\vec{r})-2\hat{n}\times\int_{\vec{r}'\varepsilon S}\left(J(\vec{r}')\times\nabla'\Psi\right)d\vec{r}' \tag{4.2.1.2}$$

where

$$\Psi(\vec{r})=\frac{e^{ikR}}{4\pi R} \tag{4.2.1.3}$$

is the Green's function for the incoming plane wave.

The Method of Moments (MoM) approach to solving the EFIE or the MFIE is to discretize the equation by expanding $J(\vec{r})$ in terms of a set of basis functions $B_n(\vec{r})$ as follows

$$J(\vec{r})=\sum_{n=1}^{N}j_n B_n(\vec{r}) \tag{4.2.1.4}$$

where $J = \{j_n\}$ the vector of expansion coefficients. When this expansion is substituted into the integral equation (4.2.1.1) and the result multiplied by a basis function and integrated over the scattering surface, the problem reduces to solving a linear system of equations

$$Z\cdot J=V \tag{4.2.1.5}$$

for the vector of expansion coefficients $J = \{j_n\}$, where the entries in the matrix $Z = [Z_{mn}]$ and the vector $V = \{v_m\}$ are complicated double integrals over the scattering surface and must be calculated numerically. For example, the entries in $Z = [Z_{mn}]$ are given by

$$Z_{mn} = \int_{\bar{r}\varepsilon S} B_m(\bar{r}) \cdot \left\{ \frac{1}{i\omega\varepsilon} \hat{n} \times \int_{\bar{r}'\varepsilon S} \left(-\omega^2 \mu\varepsilon B_n(\bar{r}')\Psi + (\nabla' \cdot B_n(\bar{r}'))\nabla'\Psi \right) d\bar{r}' \right\} d\bar{r} \qquad (4.2.1.6)$$

Generally, N increases as the square of the frequency, and for typical problems, N is greater than 10,000. In traditional MoM algorithms, which first appeared in the late 1960's, the dense linear system $Z \bullet J = V$ is solved by a direct linear equation solution algorithm, which may be composed as an in-core or out-of-core solver. On modern parallel computers, the direct solvers may be extended to work on shared or distributed memory computer architectures.

The advantage of MoM algorithms is that they are exact representations of Maxwell's equations and highly accurate simulations are possible. The disadvantage of the traditional MoM algorithms is that the methods are computationally intensive, especially as the frequency goes up. The computational complexity of traditional MoM algorithms includes $O(N^2)$ integral evaluations to compute the matrix Z and $O(N^3)$ arithmetic operations to solve the system $Z \bullet J = V$ for J. The memory requirement for traditional MoM algorithms is $O(N^2)$. For these reasons, the traditional MoM algorithms are generally used only for low frequency problems. Although traditional MoM algorithms have been highly optimized on a variety of high-performance computing machines, the largest problems solved so far are for N on the order of 100,000.

Recently, new fast MoM algorithms based on fast, iterative linear equation solvers have been introduced. The iterative solvers rely on numerically stable and rapidly converging iteration procedures, such as the preconditioned GMRES method [Saad]. Fast matrix-vector multiply algorithms are used to compute products of the form used in the iterative procedure. The computational complexity of the fast MoM algorithms is $O(N\log N)$. The memory requirement for the fast MoM algorithms is $O(N)$. This is a remarkable reduction from the $O(N^3)$ computational complexity of the traditional MoM algorithms, and potentially allows the solution of much larger problems at higher frequencies.

Rohklin [Rohklin-1, Rohklin-2] has introduced new fast MoM algorithms for the Helmholtz equation, which use iterative linear equation solvers and the fast multipole method (FMM) for fast matrix-vector multiplies. To compute products of the form $Z \bullet X$, the Z matrix is not formed or stored, rather the product $Z \bullet X$ is viewed as a field and approximately evaluated by the FMM. The mathematical formulation of the FMM is based on the theory of multipole expansions, and involves translation (change of center) of multipole expansions and spherical harmonic filtering. The computational complexity of these new methods is $O(N\log N)$ and the memory requirement is $O(N)$.

Building on the FMM approach, Dembart, Epton and Yip [Dembart-1 to 4] at Boeing have implemented a fast MoM algorithm in a production grade electromagnetics code used by the company for radar cross-section (RCS) studies. Problems for which the number of unknowns is on the order of 10,000,000 have been solved with this code. Boeing's fast solver uses the preconditioned GMRES iterative method, which requires only the calculation of products of the form $Z \bullet X$, combined with a multilevel FMM for fast matrix-vector multiplies. The solver may be summarized as follows:

1. The preconditioned GMRES iterative solution method is used to solve the system $Z \bullet J = V$. This method does not require computation and storage of the matrix Z, but rather requires the capability to compute, for a given vector X, the product $Z \bullet X$.

2. To compute the product $Z \bullet X$, the Z matrix is first decomposed into two pieces which represent contributions from "close together" and "well separated" basis functions, respectively

$$Z = Z_{near} + Z_{far} \tag{4.2.1.7}$$

3. The matrix Z_{near} is sparse, and it is computed once and for all, directly from the integral representation for its entries, in $O(N)$ integral evaluations. The product $Z_{near} \bullet X$ is computed directly for each X in $O(N)$ arithmetic operations.

4. The matrix Z_{far} is dense, but it is never computed at all—rather the product $Z_{far} \bullet X$ is computed for each X by a multilevel FMM in $O(N\log N)$ arithmetic operations.

Boeing's multilevel FMM method is formulated by enclosing the scatterer in a cube and then successively refining the cube into subcubes until the dimensions of the finest cubes are on the order of several wavelengths. At each level in the cube hierarchy two multipole expansions are computed: the outer expansion (field outside the cube due to sources inside the cube) and the inner expansion (field inside the cube due to sources outside the cube). The outer and inner expansions are efficiently represented by far-field signature functions. The key computations are translating multipole expansions (change of center) and harmonic analysis/synthesis of signature functions. The multilevel FMM calculation begins by computing the outer expansion at the finest level from X. Next, the outer-to-outer translation operation is applied to traverse up the cube hierarchy computing the outer expansions at all levels. Next, the outer-to-inner and inner-to-inner translations are used to traverse down the cube hierarchy computing the inner expansion at all levels. Finally, the matrix-vector product $Z_{far} \bullet X$ is then computed from the inner expansion at the finest level.

The $O(N\log N)$ computational complexity of the FMM results from the computational strategy of applying the outer-to-inner translation at each of the cube hierarchy as follows. At the coarsest level in the cube hierarchy, the outer-to-inner translation is applied to all pairs of cubes that are non-adjacent. For all finer levels in the cube hierarchy, the outer-to-inner translation is applied only to pairs of cubes that are non-adjacent at the given level, and whose parents are adjacent at the next higher level.

Fast MoM algorithms, such as Boeing's described above, have the potential to compute the electromagnetic scattering from complex objects at frequencies 10 to 100 times higher than possible with traditional MoM algorithms. The ultimate potential of these fast MoM algorithms is limited by two memory-related bottlenecks: low reuse of data and non-unit stride. For these reasons, we have chosen to base the *Method of Moments Benchmark* on Boeing's fast solver. The benchmark computes the far-field component of the wave field generated by a collection of radiating scalar sources. The evaluation of the far-field corresponds to the evaluation of the matrix-vector product $Z_{far} \bullet X$ described above. The computational method implemented in the benchmark is a scalar multilevel FMM similar to the multilevel FMM used in Boeing's fast solver. The key FMM kernels represented in the benchmark are the translation operations and spherical harmonic filtering. Detailed specifications of the *Method of Moments Benchmark* are given in the following sections.

4.2.1.1 Input

An input set for the *Method of Moments Benchmark*, which contains everything required to run the benchmark, consists of three binary input files: source strengths, cubes, and translation operators. The contents of the files are described in the sections below.

4.2.1.1.1 Source Strength File

The source points and their corresponding source strengths are specified in the *Source Strength* input file. The record types in the file are specified in the following table. The first record in the file is a record of type 1 containing a single integer number defining the number of source points. This is followed by a record of type 2 for each source point containing four floating point numbers (double) defining the three spatial coordinates of the source point and the strength of the source at the source point.

Record Type	Contents			
1	integer N			
2	double float X	double float Y	double float Z	double float Q

4.2.1.1.2 Cube File

The cube hierarchy on which the multilevel FMM operates is specified in the *Cube* input file. The record types in the file are specified in the following table. The first record in the file is a record of type 1 containing a single integer number defining the number of levels in the cube hierarchy. For each level in the cube hierarchy, there is a set of records defining the cubes in the level. The first record for a level is a record of type 1 containing a single integer number defining the number of cubes for the level. This is followed by a set of records for each cube. The first record for a cube is a record of type 2 containing three floating point numbers (double) defining the three spatial coordinates of the cube's center and an integer number defining the number of cubes in the level adjacent to the cube. This is followed by a record of type 1 for each adjacent cube containing a single integer number defining the number of the adjacent cube.

Record Type	Contents			
1	integer N			
2	double float X	double float Y	double float Z	integer M

4.2.1.1.3 Translation Operators

The tables defining the translation operators are specified in the *Translation Operator* file. The record types in the file are specified in the following table. The file contains four tables: outer-to-outer operator, inner-to-inner operator, outer-to-inner at top level, and outer-to-inner at levels below top level. The four tables all have the same structure: a list of complex numbers followed by a list of integer numbers.

The first record in the table is of type 1, containing a single integer number defining the number of complex numbers in the table. This is followed by a record of type 2, containing a single complex number (double), for each entry in the list. The next record in the table is a record of type 1, containing a single integer number defining the number of integer numbers in the table. This is followed by a record of type 1, containing a single integer, for each entry in the list.

Record Type	Contents
1	integer N
2	double complex X

4.2.1.2 Algorithmic Specification

The *Method of Moments Benchmark* computes the far-field component of the wave field generated by a collection of radiating scalar sources. The computational method implemented in the benchmark is a scalar multilevel FMM similar to the multilevel FMM used by Boeing in its fast MOM solver discussed above. The mathematical formulation of the scalar multilevel FMM relies on the theory of scalar multipole expansions and the theory of harmonic analysis/synthesis of signature functions. We present the specifications of the benchmark in the following order.

5. Field Generated by a Distribution of Scalar Sources
6. Multilevel Fast Multipole Method
7. Translation Operations
8. Spherical Harmonic Synthesis/Analysis

4.2.1.2.1 Field Generated by a Distribution of Scalar Sources

For the *Method of Moments Benchmark,* we define a scalar wave field as a field $\phi\left(\vec{x}\right)$ which satisfies the scalar Helmholtz equation

$$\left(\nabla^2 + k^2\right)\phi = 0 \tag{4.2.1.8}$$

where k is the wave number. The benchmark computes the field generated by a collection of sources radiating from source points $\{\vec{x}_i\}_{i=1}^{N}$ with amplitudes $\{q_i\}_{i=1}^{N}$ at a collection of field points $\{y_i\}_{j=1}^{M}$. The value of the field at the field points is given by

$$\phi\left(\vec{y}_j\right) = \sum_{i=1}^{N} q_\alpha h_0\left(k\left|\vec{y}_j - \vec{x}_i\right|\right) \tag{4.2.1.9}$$

where h_o is the spherical Bessel function of the first kind (we assume a time variation of $e^{-i\omega t}$).

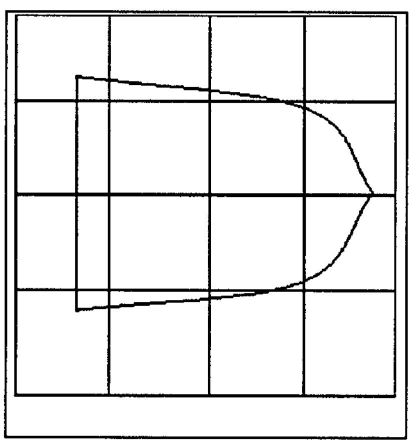


Figure 4-1: Coarse Grid and Scatterer

For convenience in the *Method of Moments Benchmark*, we take the field points to be identical to the source points. Under this assumption, the computational complexity of computing the field values at the field points by approximately evaluating the Bessel functions and doing the sum is $O(N^2)$. This is similar to the computational complexity of the matrix-vector multiply step in an iterative MoM solver for N unknowns.

4.2.1.2.2 Multilevel Fast Multipole Method

In this section we formulate the multilevel fast multipole method and show that the computational complexity of the method is $O(N\log N)$.

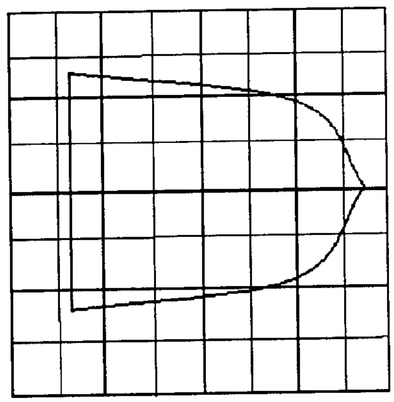


Figure 4-2: Fine Grid and Scatterer

We begin by introducing the multilevel cubical grid system utilized in the multilevel FMM. The source and field points are enclosed in a cube. This cube is then subdivided into eight equal subcubes, and each of those is similarly sub-divided, until a sufficiently fine grid is achieved. The dimensions of the cubes at the finest level are on the order of several wavelengths and the number of levels is $O(N\log N)$. At each level of this grid system there is a set of "active" cubes, that is, cubes which contain the source or field points. Only the active cubes are used in the multilevel FMM. One level of such a grid is depicted in Figure 4-1. A second, finer grid is shown in Figure 4-2. To simplify the figures the illustrations are two-dimensional. Cubes are represented as squares and the scatterer as a one-dimensional curve. The source and field points are distributed along the scatterer and are not explicitly shown in the figures. To simplify the description of the multilevel FMM, we will describe the method for the two level (fine and coarse) cubical grid system, shown in Figure 4-1 and Figure 4-2. As appropriate, we will indicate which steps must be repeated for cube hierarchies with more than two levels.

First, we introduce the outer-to-inner translation. Referring to the fine grid shown in Figure 4-2, we consider two distinct cubes Cd and Ce with centers $\vec{d}$ and $\vec{e}$, respectively. The finite degree outer expansion centered at $\vec{d}$ for the wave field $\phi_d^{(d)}(\vec{x})$ outside Cd due to the sources inside Cd has the form

$$\phi_d^{(d)}(\vec{x}) = \sum_{n=0}^{N_d} \sum_{m=-n}^{n} D_n^m h_n\left(k\left|\vec{x}-\vec{d}\right|\right) Y_n^m\left(\left(\vec{x}-\vec{d}\right)/\left|\vec{x}-\vec{d}\right|\right) \tag{4.2.1.10}$$

where the coefficients D_n^m are given by

$$D_n^m = \sum_{i=1}^{N} q_i 4\pi j_n\left(k\left|\vec{x}_i - \vec{c}\right|\right) Y_n^{-m}\left(\left(\vec{x}_i - \vec{c}\right)/\left|\vec{x}_i - \vec{c}\right|\right) \tag{4.2.1.11}$$

The outer-to-inner translation from $\vec{d}$ to $\vec{e}$ is the construction of an inner expansion for $\phi_d^{(d)}(\vec{x})$ centered at $\vec{e}$ having the form

$$\phi_{\vec{e}}^{(e)}(\vec{x}) = \sum_{n=0}^{N_e} \sum_{m=-n}^{n} E_n^m j_n\left(k\left|\vec{x} - \vec{e}\right|\right) Y_n^m\left(\left(\vec{x} - \vec{e}\right)/\left|\vec{x} - \vec{e}\right|\right) \tag{4.2.1.12}$$

that is valid for all $\vec{x}$ inside Ce. The calculation of the coefficients E_n^m from the coefficients D_n^m is described in the discussion of the translation operations below.

For a given decomposition into cubes and specified values for Nd and Ne, the accuracy of the outer-to-inner translation from $\vec{d}$ to $\vec{e}$ depends only on the distance $\vec{d} - \vec{e}$ between the cubes. For a specified accuracy, we say two cubes satisfy the far-field condition if they are sufficiently separated so that the accuracy of the outer-to-inner translation for the pair satisfies the specified accuracy.

The outer-to-inner translation provides a (1-level) computational tool to compute the field at the field points. The calculation proceeds in several steps

- For all cubes Ce we apply, for all cubes Cd that satisfy the far-field condition, the outer-to-inner translation to translate Cd's outer expansion to an inner expansion at Ce's center and sum the translated inner expansions. The result is the inner expansion for all cubes Ce due to all the sources in cubes that satisfy the far-field condition with respect to Ce.

- For all cubes Ce, we compute the field at field points in Ce as the sum of the far-field, which is given by the inner expansion computed in step1, and the near-field due to all sources inside cubes (including Ce) that don't satisfy the far-field condition with respect to Ce.

If we decompose the domain into very fine cubes, the computational complexity of the second step is reduced to only $O(1)$, but the computational complexity of the first step remains $O(N_2)$. Similarly, if we decompose the domain into course cubes, the computational complexity of the first step is reduced to only $O(1)$, but the computational complexity of the second step remains $O(N_2)$. This problem is resolved by the multilevel FMM. To specify the multilevel FMM we introduce the outer-to-outer and inner-to-inner translations.

Referring to Figure 4-1 and Figure 4-2, we consider a cube Cd at the coarse level with center $\vec{d}$ and a cube Cc at the fine level with center $\vec{c}$. The finite degree outer expansion centered at $\vec{c}$ for the wave field $\phi_c^{(c)}(\vec{x})$ outside Cc due to the sources inside Cc has the form

$$\phi_{\vec{c}}^{(c)}(\vec{x}) = \sum_{n=0}^{N_c} \sum_{m=-n}^{n} C_n^m h_n\left(k\left|\vec{x} - \vec{c}\right|\right) Y_n^m\left(\left(\vec{x} - \vec{c}\right)/\left|\vec{x} - \vec{c}\right|\right) \tag{4.2.1.13}$$

where the coefficients C_n^m are given by

$$C_n^m = \sum_{i=1}^{N} q_i 4\pi j_n\left(k\left|\vec{x}_i - \vec{c}\right|\right) Y_n^{-m}\left(\left(\vec{x}_i - \vec{c}\right) / \left|\vec{x}_i - \vec{c}\right|\right) \quad (4.2.1.14)$$

The outer-to-outer translation from $\vec{c}$ to $\vec{d}$ is the construction of an outer expansion for $\phi_c^{(c)}(\vec{x})$ centered at $\vec{d}$ having the form

$$\phi_{\vec{d}}^{(d)}(\vec{x}) = \sum_{n=0}^{N_d} \sum_{m=-n}^{n} D_n^m h_n\left(k\left|\vec{x} - \vec{d}\right|\right) Y_n^m\left(\left(\vec{x} - \vec{d}\right) / \left|\vec{x} - \vec{d}\right|\right) \quad (4.2.1.15)$$

that is valid for all $\vec{x}$ outside Cd. The calculation of the coefficients D_n^m from the coefficients C_n^m is described in the discussion of the translation operations below.

Referring again to Figure 4-1 and Figure 4-2, we consider a cube Ce at the coarse level with center $\vec{e}$ and a cube Cf at the fine level with center $\vec{f}$. Suppose that we have constructed a finite degree inner expansion centered at $\vec{e}$ for the wave field $\phi_{\vec{e}}^{(e)}(\vec{x})$ inside Ce due to the sources outside Ce of the form

$$\phi_{\vec{e}}^{(e)}(\vec{x}) = \sum_{n=0}^{N_e} \sum_{m=-n}^{n} E_n^m j_n\left(k\left|\vec{x} - \vec{e}\right|\right) Y_n^m\left(\left(\vec{x} - \vec{e}\right) / \left|\vec{x} - \vec{e}\right|\right) \quad (4.2.1.16)$$

The inner-to-inner translation from $\vec{e}$ to $\vec{f}$ is the construction of an inner expansion for $\phi_{\vec{e}}^{(e)}(\vec{x})$ centered at $\vec{f}$ having the form

$$\phi_{\vec{f}}^{(f)}(\vec{x}) = \sum_{n=0}^{N_f} \sum_{m=-n}^{n} F_n^m j_n\left(k\left|\vec{x} - \vec{f}\right|\right) Y_n^m\left(\left(\vec{x} - \vec{f}\right) / \left|\vec{x} - \vec{f}\right|\right) \quad (4.2.1.17)$$

that is valid for all $\vec{x}$ in Cf. The calculation of the coefficients F_n^m from the coefficients E_n^m is described in the discussion of the translation operations below.

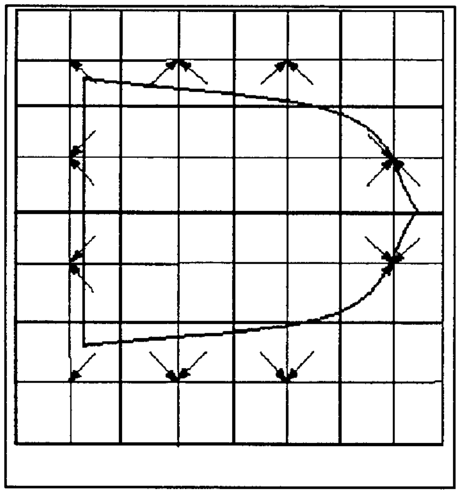


Figure 4-3: Outer-to-Outer Translation from Fine to Coarse Level

The outer-to-outer and inner-to-inner translations provide the computational tools we need to use the cube hierarchy to efficiently compute the field at the field points. The calculation proceeds in several steps, traversing up and down the cube hierarchy.

9. For all cubes Cc at the fine level we compute the finite degree outer expansion for Cc from the sources at the source points.
10. For all cubes Cd at the coarse level, we apply the outer-to-outer translation to translate the outer expansion for each of Cd's children to Cd's center and sum the translated outer expansions. The result is that we have constructed the outer expansion for all cubes Cd at the coarse level. The outer-to-outer translation from the coarse level to the fine level is shown in Figure 4-3.

For all cubes Ce at the coarse level we apply, for all cubes Cd at the coarse level that satisfy the far field condition, the outer-to-inner translation to translate Cd's outer expansion to an inner expansion at Ce's center and sum the translated inner expansions. The result is the inner expansion for all cubes Ce at the coarse level due to all the sources in cubes at the coarse level that satisfy the far-field condition with respect to Cf. The outer-to-inner translation at the coarse level is shown in Figure 4-4.

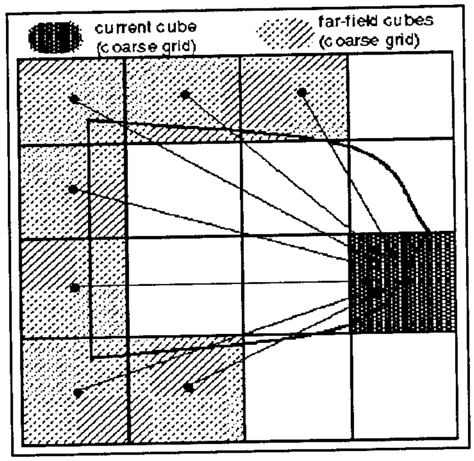


Figure 4-4: Outer-to-Inner Translation at the Coarse Level

For all cubes Ce at the coarse level, we apply, for each child Cf of Ce the inner-to-inner translation to translate the inner expansion for Ce to Cf's center. The result is the inner expansion for all cubes Cf at the fine level due to sources in cubes Cc at the fine level such that the parents of Cf and Cc satisfy the far-field condition. The inner-to-inner translation from the fine level to the coarse level is shown in Figure 4-5.

For all cubes Cf at the fine level we apply, for all cubes Cc at the fine level that satisfy the far field condition *and for which the parents of Cf and Cc don't satisfy the far-field condition*, the outer-to-inner translation to translate Cc's outer expansion to an inner expansion at Cf's center and sum the translated inner expansions. The result is the inner expansion for all cubes Cf at the fine level due to all the sources in cubes at the fine level that satisfy the far-field condition with respect to Cf. The outer-to-inner translation at the fine level is shown in Figure 4-7.

For all cubes Cf at the fine level, we compute the field at field points in Cf as the sum of the far-field, which is given by the inner expansion computed in step 5, and the near-field due to all sources inside cubes (including Cf) that don't satisfy the far-field condition with respect to Cf. The evaluation of the near-field is shown in Figure 4-6.

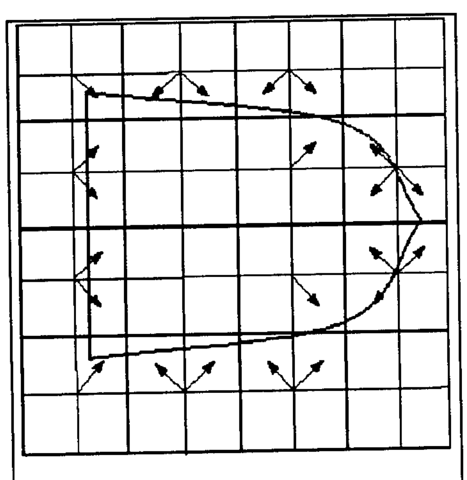


Figure 4-5: Inner-to-Inner Translation from Fine Level to Coarse Level

The two-level computation described above is easily extended to a multilevel cube hierarchy. The efficiency of the multilevel calculation derives from step 5, where the outer-to-inner translation only need to be applied at a level to cube pairs at the level for which the outer-to-inner translation was not already accounted for at the higher levels. As a result the computational cost of the translation operations is roughly the same at all levels of the cube hierarchy. Also, by using approximately log N levels the computational cost of the step 6 is roughly the same for all field points. Thus, the computational complexity of the multilevel FMM is $O(N\log N)$.

4.2.1.2.3 Translation Operations

In this section we use Rohklin's translation theorems to specify the translation operations: outer-to-inner, outer-to-outer and inner-to-inner. The theorems are expressed in terms of far field signature functions on the unit sphere. For a scalar wave field as a field $\phi(\vec{x})$, we define the associated far field signature function $\hat{\phi}(\vec{s})$ as follows

$$\hat{\phi}_{\vec{c}}(\vec{s}) = \lim_{\tau \to \infty} \left[k\tau e^{-ik\tau} \phi(\vec{c} + \tau\vec{s}) \right] \tag{4.2.1.18}$$

for points $\vec{s}$ on the unit sphere. When $\phi(\vec{x})$ is defined by a multipole expansion, we may use the spherical harmonic transform to evaluate the associated far field signature function. Accordingly, expansions $\phi_c^{(c)}(\vec{x})$ and $\phi_d^{(d)}(\vec{x})$ and the finite degree inner expansions signature functions are given by

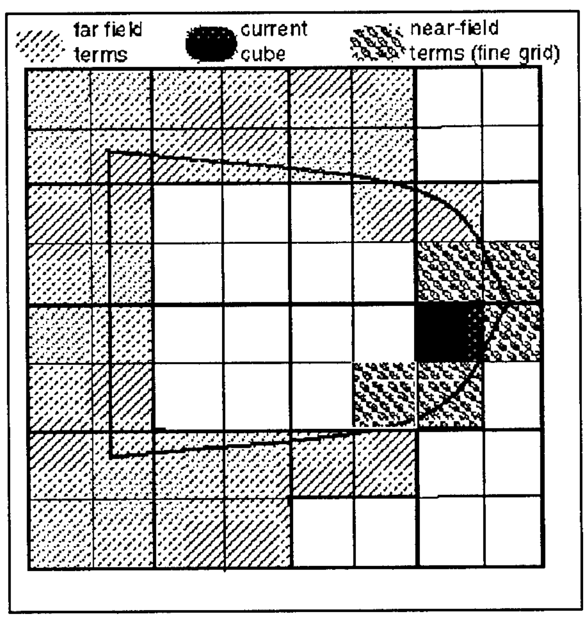


Figure 4-6: Near-field Contributions at Fine Level

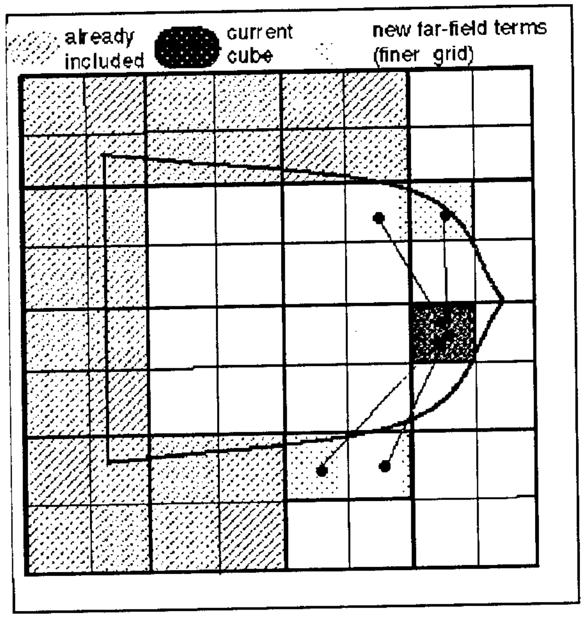


Figure 4-7: Outer-to-Inner Translation at Fine Level

$$\phi_{\vec{c}}^{(c)}(\vec{s}) = \sum_{n=0}^{N_c} \sum_{m=-n}^{n} C_n^m Z_n^n(\vec{s}) / i^{n+1} \tag{4.2.1.19}$$

$$\phi_{\vec{d}}^{(d)}(\vec{s}) = \sum_{n=0}^{Nd} \sum_{m=-n}^{n} D_n^m Z_n^n(\vec{s}) / i^{n+1} \tag{4.2.1.20}$$

$$\phi_{\vec{e}}^{(e)}(\vec{s}) = \sum_{n=0}^{Ne} \sum_{m=-n}^{n} E_n^m Z_n^n(\vec{s}) / i^{n+1} \tag{4.2.1.21}$$

$$\phi_{\vec{f}}^{(f)}(\vec{s}) = \sum_{n=0}^{N_f} \sum_{m=-n}^{n} F_n^m Z_n^n(\vec{s}) / i^{n+1} \tag{4.2.1.22}$$

Rohklin's translation theorems also make use of three operators defined on the unit sphere: the harmonic projection operator Π_N, the outer-to-outer/inner-to-inner translation operator $G(\vec{s};\vec{r},N)$ and the outer-to-inner translation operator $M(\vec{s};\vec{r},N)$.

The spherical harmonic projection operator Π_N is defined by

$$\Pi_N[f(\vec{s})] = \frac{1}{4\pi} \iint_{S_1} \delta_N(\vec{s}\cdot\vec{t}) f(\vec{t}) \partial\omega_{\vec{t}} \tag{4.2.1.23}$$

where

$$\delta_N(\vec{s}\cdot\vec{t}) = \sum_{0}^{N}(2n+1)P_n(\vec{s}\cdot\vec{t}) = \sum_{n=0}^{N}\sum_{m=-n}^{n}4\pi Z_n^m(\vec{s})\left[Z_n^m(\vec{t})\right]^* \tag{4.2.1.24}$$

The outer-to-outer/inner-to-inner translation operator $G(\vec{s};\vec{r},N)$ is defined by

$$G(\vec{s};\vec{r},N) = \sum_{n=0}^{N}(2n+1)i^n j_n(k|\vec{r}|)P_n(\vec{s}\cdot\vec{t}) \tag{4.2.1.25}$$

$$= \sum_{n=0}^{N}\sum_{m=-n}^{n}4\pi i^n j_n(k|\vec{r}|)Z_n^m(\vec{s})(-1)^m Z_n^{-m}(\vec{r}) \tag{4.2.1.26}$$

The outer-to-inner translation operator $M(\vec{s};\vec{r},N)$ is defined by

$$M(\vec{s};\vec{r},N) = \sum_{n=0}^{N}(2n+1)i^n h_n(k|\vec{r}|)P_n(\vec{s}\cdot\vec{t}) \tag{4.2.1.27}$$

$$= \sum_{n=0}^{N}\sum_{m=-n}^{n}4\pi i^n h_n(k|\vec{r}|)Z_n^m(\vec{s})(-1)^m Z_n^{-m}(\vec{r}) \tag{4.2.1.28}$$

With these definitions in place, Rohklin's translation theorems may be states as follows:

Outer-to-Outer Translation Theorem. The finite degree outer signature functions $\hat{\phi}_{\vec{c}}^{(c)}(\vec{s})$ and $\hat{\phi}_{\vec{d}}^{(d)}(\vec{s})$ are related to one another by the identity

$$\hat{\phi}_{\vec{d}}^{(d)}(\vec{s}) = \Pi_{N_d}\left[G(\vec{s};\vec{d}-\vec{c},N_c+N_d)\hat{\phi}_{\vec{c}}^{(c)}(\vec{s})\right] \tag{4.2.1.29}$$

Outer-to-Inner Translation Theorem. The finite degree outer signature function $\hat{\phi}_{\vec{d}}^{(d)}(\vec{s})$ and the finite degree inner signature function $\hat{\phi}_{\vec{e}}^{(e)}(\vec{s})$ are related to one another by the identity

$$\hat{\phi}_{\vec{e}}^{(e)}(\vec{s}) = \Pi_{N_e}\left[G(\vec{s};\vec{e}-\vec{d},N_d+N_e)\hat{\phi}_{\vec{d}}^{(d)}(\vec{s})\right] \tag{4.2.1.30}$$

Inner-to-Inner Translation Theorem. The finite degree inner signature functions $\hat{\phi}_{\vec{e}}^{(e)}(\vec{s})$ and $\hat{\phi}_{\vec{f}}^{(f)}(\vec{s})$ are related to one another by the identity

$$\hat{\phi}_{\vec{f}}^{(f)}(\vec{s}) = \Pi_{N_f}\left[G(\vec{s};\vec{f}-\vec{e},N_e+N_f)\hat{\phi}_{\vec{e}}^{(e)}(\vec{s})\right] \tag{4.2.1.31}$$

The importance of Rohklin's translation theorems is that they reduce translation of multipole expansions to pointwise multiplication on the unit sphere of a signature function by a translation operator, followed by filtering.

4.2.1.2.4 Spherical Harmonic Synthesis/Analysis

In this section we specify Epton and Dembart's method for filtering and interpolating signature functions. Filtering of signature functions is needed for implementation of spherical harmonic projection operator Π_N used in Rohklin's translation theorems stated above. Interpolation and filtering of signature functions is needed to move discrete tabulations of the functions on the unit sphere between levels without losing accuracy. First, we define spherical harmonic analysis and synthesis for finite degree signature functions tabulated on a grid on the unit sphere. Then we specify Epton and Dembart's algorithm for performing spherical harmonic analysis and synthesis. Finally, we describe Epton and Dembart's method for filtering and interpolating signature functions.

For a finite degree signature function $f(\vec{s})$ of the form

$$f(\vec{s})=\sum_{n=0}^{N}\sum_{m=-n}^{n} f_n^m Z_n^m(\vec{s}) \tag{4.2.1.32}$$

a tabulation $\{f(\theta_k,\phi_l): 0 ? \ k \le N_\theta -1, 0 \le l \le N_\phi -1\}$ of $f(\vec{s})$ on a grid of $\vec{s}$ values on the unit sphere defined as follows (N_θ and N_ϕ will be specified in terms of N.)

$$\Theta = \{\theta_k : \theta_k = (k+1/2)\pi / N_\theta, k = 0; N_\theta - 1\} \tag{4.2.1.33}$$

$$\Phi = \{\phi_l : \phi_l = 2\pi l / N_\phi, l = 0; N_l - 1\} \tag{4.2.1.34}$$

The process of obtaining the coefficients $\{f_n^m : 0 \le n \le N, |m| \le n\}$ from the tabulation $\{f(\theta_k,\phi_l): 0 ? \ k \le N_\theta -1, 0 \le l \le N_\phi -1\}$ is called spherical harmonic analysis. The inverse process of obtaining the tabulation $\{f(\theta_k,\phi_l): 0 ? \ k \le N_\theta -1, 0 \le l \le N_\phi -1\}$ from the coefficients $\{f_n^m : 0 \le n \le N, |m| \le n\}$ is called spherical harmonic synthesis.

Epton and Dembart's algorithm for performing spherical harmonic analysis and synthesis is based on the observation that the spherical harmonics $Z_n^m(\vec{s})$ can be viewed as trigonometric polynomials in θ and ϕ as follows

$$f(\vec{s})=\sum_{n=0}^{N}\sum_{m=-n}^{n} f_n^m Z_n^m(\vec{s})=\sum_{n=0}^{N}\sum_{m=-n} f_n^m z_n^m(\theta)e^{im\phi} = f(\theta,\phi) \tag{4.2.1.35}$$

where $\vec{s}$ ..(θ,ϕ) and $Z_n^m(\vec{s}) = z_n^m(\theta)e^{im\phi}$. Changing the order of summation gives

$$f(\theta,\phi)=\sum_{m=-N}^{N}\sum_{n=|m|}^{N}f_n^m z_n^m(\theta)e^{im\phi}=\sum_{m=-N}^{N}f^{(m)}(\theta)e^{im\phi} \tag{4.2.1.36}$$

where the functions $\{f^{(m)}(\theta):-N\le m\le N\}$ are given by

$$f^{(m)}(\theta)=\sum_{n=|m|}^{N}f_n^m z_n^m(\theta) \tag{4.2.1.37}$$

A linear system of equations relating the f_n^m and the $f(\theta_k,\phi_l)$ is formulated by comparing discrete and analytic expansions of $f^{(m)}(\theta)$ as follows.

Application of the discrete Fourier inversion theorem to $f(\theta,\phi)$ gives $f^{(m)}(\theta)$ as follows

$$f^{(m)}(\theta)=\frac{1}{N_\phi}\sum_{l=0}^{N_\phi-1}e^{im\phi_l}f(\theta,\phi_l) \tag{4.2.1.38}$$

Application of the shifted cosine transform to $f^{(m)}(\theta)$ for m even gives

$$f^{(m)}(\theta)=\sum_{p=0}^{N_\theta-1}\cos[p\theta]F_p^{(m)}\cdots m=even \tag{4.2.1.39}$$

where the numbers $\{F_p^{(m)}:m=even,-N\le m\le N,0\le p\le N_\theta-1\}$ are given by

$$F_p^{(m)}=\sum_{k=0}^{N_\theta-1}\frac{1}{\cos[p\theta]}\left\{\frac{1}{N_\phi}\sum_{l=0}^{N_\phi-1}e^{im\phi_l}f(\theta_k,\phi_l)\right\}\cdots m=even \tag{4.2.1.40}$$

Similarly, application of the shifted sine transform to $f^{(m)}(\theta)$ for m odd gives

$$f^{(m)}(\theta)=\sum_{p=0}^{N_\theta-1}\sin[(p+1)\theta]F_p^{(m)}\cdots m=odd \tag{4.2.1.41}$$

where the numbers $\{F_p^{(m)}:m=odd,-N\le m\le N,0\le p\le N_\theta-1\}$ are given by

$$F_p^{(m)}=\sum_{k=0}^{N_\theta-1}\frac{1}{\sin[(p+1)\theta]}\left\{\frac{1}{N_\phi}\sum_{l=0}^{N_\phi-1}e^{im\phi_l}f(\theta_k,\phi_l)\right\}\cdots m=odd \tag{4.2.1.42}$$

Expanding $z_n^m(\theta)$ in a cosine series for m even gives

$$f^{(m)}(\theta)=\sum_{n=|m|}^{N}f_n^m z_n^m(\theta)=\sum_{n=|m|}^{N}f_n^m\sum_{p=0}^{n}A_{n,p}^m\cos[p\theta]\cdots m=even \tag{4.2.1.43}$$

$$= \sum_{p=0}^{N} \left\{ \sum_{\substack{n=|m| \\ n \geq p}}^{N} A_{n,p}^{m} f_{n}^{m} \right\} \cos[p\theta] \cdots m = even \tag{4.2.1.44}$$

Similarly, expanding $z_n^m(\theta)$ in a sine series for m odd, gives

$$f^{(m)}(\theta) = \sum_{n=|m|}^{N} f_n^m z_n^m(\theta) = \sum_{n=|m|}^{N} f_n^m \sum_{p=0}^{n-1} A_{n,p+1}^{m} \sin[(p+1)\theta] \cdots m = odd \tag{4.2.1.45}$$

$$= \sum_{p=0}^{N-1} \left\{ \sum_{\substack{n=|m| \\ n \geq p+1}}^{N} A_{n,p+1}^{m} f_{n}^{m} \right\} \sin[(p+1)\theta] \cdots m = odd \tag{4.2.1.46}$$

By comparing the discrete and analytic cosine expansions of $f^{(m)}(\theta)$ for m even, we obtain the following system of linear equations

$$\sum_{\substack{n=|m| \\ n \geq p}}^{N} A_{n,p}^{m} f_{n}^{m} = F_{p}^{(m)} \cdots m = even \tag{4.2.1.47}$$

Similarly, by comparing the discrete and analytic sine expansions of $f^{(m)}(\theta)$ for m odd, we obtain the following system of linear equations

$$\sum_{\substack{n=|m| \\ n \geq p+1}}^{N} A_{n,p+1}^{m} f_{n}^{m} = F_{p}^{(m)} \cdots m = odd \tag{4.2.1.48}$$

Epton and Dembart's method for filtering and interpolating signature functions consists of the following computational steps to transform an input tabulation $f(\theta_k, \phi_l)$ of a finite degree signature function $f(\vec{s})$ on the unit sphere to an output tabulation $g(\theta_k, \phi_l)$ of the filtered/interpolated finite degree signature function $g(\vec{s})$ on the unit sphere.

Step	Action
1	Starting with a tabulation $f(\theta_k, \phi_l)$ of a finite degree signature function $f(\vec{s})$ on the unit sphere, perform the discrete Fourier transform in the ϕ direction using an FFT.
2	Split the transformed data into even and odd frequency data arrays.
3	Transpose the even and odd frequency data arrays.
4	Apply the discrete shifted cosine transform to the even frequency data array in the θ direction using a FFT. Similarly, apply the discrete shifted sine transform to the odd frequency data array in the θ direction using a FFT.
5	Solve the linear system of equations defined by equations 4.2.1-46 and 4.2.1-47 to compute f_n^m.

6	Compute g_n^m from f_n^m. For filtering, g_n^m is obtained from f_n^m by dropping terms. For interpolation, g_n^m is obtained from f_n^m by adding zero terms.
7	Apply the inverse discrete shifted cosine transform to the even frequency data array in the θ direction using a FFT. Similarly, apply the inverse discrete shifted sine transform to the odd frequency data array in the θ direction using a FFT.
8	Transpose the even and odd frequency data arrays.
9	Combine the transformed even and odd frequency data into a single data array.
10	Perform the inverse discrete Fourier transform in the ϕ direction using an FFT to compute the tabulation $g(\theta_k, \phi_l)$ of the finite degree signature function $g(\vec{s})$ on the unit sphere.

4.2.1.3 Output

An output set for the *Method of Moments Benchmark* consists of two binary files: computed far-field and metrics report. The contents of the files are described in the sections below.

4.2.1.3.1 Far-Field

The far-field computed at the field points, which are identical with the source points specified in the *Source Strength* file, are output to the *Far-Field* file. The record types in the file are specified in the table below. The first record in the file is a record of type 1 containing a single integer number defining the number of field points. This is followed by a record of type 2 for each field point containing three floating point numbers (double) and a single complex number (double) defining the three spatial coordinates of the field point and strength of the computed far-field at the field point.

Record Type	Contents			
1	integer N			
2	double float X	double float Y	double float Z	double complex E

4.2.1.3.2 Metrics Report

The *Method of Moments Benchmark* collects data for the evaluation of the metrics specified in Section 4.2.1.5. The metric data is output in the *Metrics Report* output file. The record types in the file are specified in the table below. The first record of the file is a record of type 1 containing three floating-point numbers (float) defining the first and secondary performance metrics. The second record of the file is a record of type 2 containing six floating-point numbers (float) defining the tertiary performance measures for the translation operations. The third record of the file is a record of type 2 containing six floating-point numbers (float) defining the tertiary performance measures for the spherical harmonic filtering/synthesis calculations.

Record Type	Contents					
1	float T1	float T2	float T3			
2	float T1	float T2	float T3	float T4	float T5	float T6

Implementers may add additional records to the *Metric Report* file to record data for evaluating their implementation relative to the metrics for scalability with respect to problem size and scalability with respect to processors, if these metrics apply to their implementation.

4.2.1.4 Acceptance Test

The *Acceptance Test* for the *Method of Moments Benchmark* consists of a comparison between the reference far-field and the computed far-field. The reference far-field is specified in the *Far-Field Reference* file. The number of digits of accuracy specified for the acceptance test is also defined in the *Far-Field Reference* file. The contents of the *Far-Field Reference* file are described below.

The computed far-field at a field point passes the acceptance test if the computed far-field and the reference far-field agree to the specified number of digits. To avoid possible self-checking errors, implementers should perform the acceptance test on a computer separate from the PIM computer that is being benchmarked. Implementers should report the results of the acceptance test in the *Acceptance Test Report* file. The contents of the *Acceptance Test Report* file are described below.

4.2.1.4.1 Far-Field Reference

The field points and their corresponding far-field strengths used in the acceptance test are specified in the *Far-Field Reference* input file. The record types in the file are specified in the table below. The first record in the file is of type 1 containing a single integer number defining the number of field points. This is followed by a record of type 2 for each field point containing three floating point numbers (double) and a single complex number (double) defining the three spatial coordinates of the field point and strength of the far-field at the field point.

Record Type	Contents			
1	integer N			
2	double float X	double float Y	double float Z	double complex E

4.2.1.4.2 Acceptance Test Report

The results of the acceptance test are output in the *Acceptance Test Report* file. The record types in the file are specified in the table below. The first record of the file is a record of type 1 containing a single integer number defining how many field points failed the acceptance test. If any of the field points failed the acceptance test, the first record is followed by a record for each field point that failed the acceptance test. The records are of type 2 containing three floating point numbers (double) and two complex numbers (double) defining the three spatial coordinates of the field point and the strengths of the reference field and computed far-field at the field point.

Record Type	Contents				
1	integer N				
2	double float X	double float Y	double float Z	dble complex E1	dble complex E2

4.2.1.5 Metrics

The *Metrics* for the *Method of Moments Benchmark* consist of three measures: performance, scalability with respect to problem size and scalability with respect to processors. The metrics are described the sections below. The performance metric is a quantitative metric measured directly by the *Method of Moments Benchmark* code and reported in the *Metric Report* file. The scalability metrics are subjective measures and should be evaluated by the implementers with respect to their PIM architecture.

4.2.1.5.1 Performance

The performance of an implementation of the *Method of Moments Benchmark* is measured in wall-clock time. The performance measures are summarized in the table below. The primary measure of performance is the total wall-clock time for the calculation of the far-field by the multilevel FMM. The secondary measure of performance is the breakdown of the total time into the total time for all translation operations and the total time for all spherical harmonic filtering/synthesis calculations. The tertiary measures of performance are the breakdown of the secondary measures into the total time for each of the six steps in the multilevel FMM as specified in Section 4.2.1.

Metric	Wall-Clock Time
primary	Total Time for Multilevel FMM
secondary	Total Time for Translation Operations
tertiary	Total Time for Translation Operations During Initialize Sig. Functions at Finest Level
tertiary	Total Time for Translation Operations During Outer-to-Outer below Coarsest Level
tertiary	Total Time for Translation Operations During Outer-to-Inner at Coarsest Level
tertiary	Total Time for Translation Operations During Inner-to-Inner above Finest Level
tertiary	Total Time for Translation Operations During Outer-to-Inner below Coarsest Level
tertiary	Total Time for Translation Operations During Evaluation of Far-Field at Finest Level
secondary	Total Time for Filtering/Synthesis
tertiary	Total Time for Filtering/Synthesis During Initialize Sig. Functions at Finest Level

tertiary	Total Time for Filtering/Synthesis During Outer-to-Outer below Coarsest Level
tertiary	Total Time for Filtering/Synthesis During Outer-to-Inner at Coarsest Level
tertiary	Total Time for Filtering/Synthesis During Inner-to-Inner above Finest Level
tertiary	Total Time for Filtering/Synthesis During Outer-to-Inner below Coarsest Level
tertiary	Total Time for Filtering/Synthesis During Evaluation of Far-Field at Finest Level

The demonstration benchmark code includes timing subroutines. Calls to the timing routines are embedded in the benchmark code to measure the specified performance metrics. The benchmark code also includes a subroutine to output the performance metrics to the *Metrics Report* file.

4.2.1.5.2 Scalability With Respect to Problem Size

The *Method of Moments Benchmark* includes a "flat plate" test series that sets the field points equal to the source points and geometrically scales the number of points. Over this set of test cases, a log/log plot of the total wall-clock time for the calculation of the far-field by the multilevel FMM (the primary performance measure) versus number of points should give approximately a straight line with slope +1. Deviations from this expected behavior give an indication of the scalability with respect to problem size of an implementation of the *Method of Moments Benchmark*. The flat plate series is described in more detain in Section 4.2.1.8.

4.2.1.5.3 Scalability With respect to Processors

The flat plate test series can also be used to study the scability of an implementation of the *Method of Moments Benchmark* with respect to processors. For a given size problem, a log/log plot of the total wall-clock time for the calculation of the far-field by the multilevel FMM (the primary performance measure) versus number of processors should give approximately a straight line with slope –1, if the implementation scales linearly with the number of processors. Deviation from the straight line indicates the extent of linear performance. Plotting the total time versus number of processors over the set of test cases in the flat plate test series gives an indication of how the scalability with respect to processor varies over problem size.

4.2.1.6 Baseline Source Code

Baseline source code is available at http://www.aaec.com/projectweb/dis.

4.2.1.7 Baseline Performance Figures

Baseline performance figures are available at http://www.aaec.com/projectweb/dis.

4.2.1.8 Test Data Sets

Test data sets are available at http://www.aaec.com/projectweb/dis.

4.2.2 Simulated SAR Ray Tracing

The Simulated SAR image process consists of three major steps:

- Geometry Sampling
- Electromagnetic Scattering Prediction
- Image Formation

Of these, the Geometry Sampling and the Image Formation steps take the majority of the CPU time, and present the most interesting sub-applications to build benchmarks around.

The Geometry Sampling is accomplished by using ray tracing to simulate the physical optics part of the electromagnetic scattering problem. Here, rays are sent out from an idealized synthetic aperture, and intersections between objects and these rays are found. At each intersection, the information about the object intersected is determined and recorded. A specular reflection ray is generated at the intersection point, and it is then fired into the object database. This creates a recursive process that allows the radar energy to be followed as it bounces through the target database. The rays are followed until a user-defined number of reflections have occurred, or until the ray leaves the database area. This process results in a linked list of intersection information that is called a *ray history*. The ray history is the output of the Geometry Sampling process.

For this benchmark, the Geometry Sampling section is further broken down into two subparts that, when put together, form the complete Geometry Sampling benchmark.

The first sub-part is a ray server. This consists of the intersection-finding part of the ray-tracing problem. It tests a bounding box structure, described in detail later, to find a list of objects that possibly intersect the ray under test. It then applies the intersection code for each type of object in this list. Once all intersections are found, it returns the intersection that is closest to the starting point of the ray under test.

The second sub-part is the Ray-Tracing Controller. This generates the grid of rays that simulate the synthetic aperture and generates the specular reflection rays based on the intersection information. It calls the ray server and passes a new ray from either the synthetic aperture grid or a reflection ray. It also tests the ray before it is sent, to see if the maximum number of reflections have been processed. This controller then takes the intersection information returned from the ray server and creates the ray history by creating the linked list for each ray fired from the synthetic aperture grid.

The Image Formation part of the simulated SAR benchmark suite takes an array of electromagnetic responses and maps them into a rectangular grid of the slant plane. This remapped EM array is then passed through a convolution that adds the effects of the system IPR. The final step is to convert the complex image into something that can be displayed. To achieve this, magnitude detection is performed on the complex image.

4.2.2.1 Input

The input for the Simulated SAR benchmark is divided into two parts: the inputs for the Ray-Tracing portion of the benchmark, and a separate set of inputs for the Image Formation portion. Each of these inputs specifications contain subsections for *Input Variables* and *Input Data*. The *Input Variables* contain the information that would be contained in a command line argument. The *Input Data* contains the databases used by each of the benchmarks.

4.2.2.1.1 Recursive Ray Tracer Input

4.2.2.1.1.1 Input Variables

Aperture specification	The aperture specification gives the location of the radar in global coordinates, a look direction vector, and a field of view (FOV) that simulates the synthetic aperture. If the FOV is given as ZERO, then a Parallel Projection is used instead of a Perspective Projection. In the Parallel case the target is centered in the Aperture and the look direction vector will give the azimuth and elevation angles need to place the aperture in the correct position.
Number of sample points	The number of sample points tells the ray-tracer the sample resolution in both range (Vertical) and cross range (Horizontal). This is directly related to the radar resolution. Three sampling resolutions will be used in this benchmark. • 512 x 512 for a small, low resolution, sample. • 2048 x 2048 for a medium sample. • 4096 x 4096 for a large sample.
Maximum number of reflections	This will specify the maximum number of reflections that are to be traced. This will be set to 3 for all benchmark runs.
Database Specifications	This will include the database name that will be used to select between the target databases specified in the following section. This specification will also contain a target rotation (yaw, pitch, and roll) and translation that positions the target in global coordinates.

4.2.2.1.1.2 Input Data

The target databases will consist of models built from both polygons and solid geometry using Constructive Solid Geometry. Each target database will also contain a hierarchical bounding-box structure. For the Polygon target model, we will assume triangular facets or three-sided polygons. The file format of each model type is described below.

4.2.2.1.1.2.1 Polygon target model file format

The overall file format is as follows.

File header

Part1

Sub-part 1
Sub-part 2
.
.
Sub-part n

Part2

Sub-part 1
Sub-part 2
.
.
Sub-part n

•
•

Part-n

Sub-part 1
Sub-part 2
.
.
Sub-part n

File header

Target model name	char[256]	ASCII test description
Model bounding box	float [3][2]	Minimum X target value
		Maximum X target value
		Minimum Y target value
		Maximum Y target value
		Minimum Z target value
		Maximum Z target value
Number of parts	int	Total number of modeled parts

Part format

Part name	char[256]	ASCII part description
Part bounding box	float[3][2]	Minimum X part value

		Maximum X part value
		Minimum Y part value
		Maximum Y part value
		Minimum Z part value
		Maximum Z part value
Number of sub-parts	int	Total number of sub-parts

Sub-part format

Sub-Part name	char[256]	ASCII subpart description
Sub-Part bounding box	float[3][2]	Minimum X part value
		Maximum X part value
		Minimum Y part value
		Maximum Y part value
		Minimum Z part value
		Maximum Z part value
Number of nodes	int	Number of facet vertices
Vertex list	float[N][3]	Number of nodes by x,y,z
		Coordinates
Number of Facets	int	Number of 3-sided facets
		Built from the above nodes
Facet list	int [M][5]	Number of facets by
		vertex 1 index into vertex list,
		vertex 2 index into vertex list, vertex 3 index into vertex list,
		Material index, and Surface index.

Here is an example file for a simple unit box with the lower left corner at (0,0,0). The material type is 1 and the surface type is 3 for all facets.

```
Box
0.0 1.0 0.0 1.0
6

Front Face
0.0 1.0 0.0 0.0 0.0 1.0
1
Front Face
0.0 1.0 0.0 0.0 0.0 1.0
4
0.0 0.0 0.0
0.0 0.0 1.0
1.0 0.0 0.0
1.0 0.0 1.0
2
1 2 4        1      3
1 3 4        1      3

Back Face
0.0 1.0 1.0 1.0 0.0 1.0
1
Back Face
0.0 1.0 1.0 1.0 0.0 1.0
4
0.0 1.0 0.0
0.0 1.0 1.0
1.0 1.0 0.0
1.0 1.0 1.0
2
1 2 4        1      3
1 3 4        1      3

Left Face
0.0 0.0 0.0 1.0 0.0 1.0
1
Left Face
0.0 0.0 0.0 1.0 0.0 1.0
4
0.0 0.0 0.0
0.0 1.0 0.0
0.0 0.0 1.0
0.0 1.0 1.0
2
1 2 4        1      3
1 3 4        1      3

Right Face
1.0 1.0 0.0 1.0 0.0 1.0
1
Right Face
1.0 1.0 0.0 1.0 0.0 1.0
4
1.0 0.0 0.0
1.0 1.0 0.0
1.0 0.0 1.0
1.0 1.0 1.0
```

```
2
1 2 4        1     3
1 3 4        1     3

Top Face
1.0 0.0 0.0 1.0 1.0 1.0
1
Top Face
0.0 0.0 0.0 1.0 1.0 1.0
4
0.0 0.0 1.0
0.0 1.0 1.0
1.0 0.0 1.0
1.0 1.0 1.0
2
1 2 4        1     3
1 3 4        1     3

Bottom Face
1.0 1.0 0.0 1.0 0.0 0.0
1
Bottom Face
1.0 1.0 0.0 1.0 0.0 0.0
4
0.0 0.0 0.0
1.0 0.0 0.0
0.0 1.0 0.0
1.0 1.0 0.0
2
1 2 4        1     3
1 3 4        1     3
```

4.2.2.1.1.2.2 CSG solid target model file format

For the CSG solid target models, we will use the BRL CAD .cg geometry description file format. This is an older format, but it is easier to work with. The geometry description file contains all the information required to define the physical components of a geometry model and the operations that are required to construct it. The format of each line of data is a detailed record format. The entire file is divided into four basic parts: title control, primitive definitions, region definitions, and region identifications.

The title control section simply provides a name for the model and the units of measure (millimeters, centimeters, meters, inches, or feet abbreviated as mm, cm, m, in, and ft).

The primitive definition section uniquely describes each of the primitive solids to be used in the model construction (See Figure 4-8).

The region definition section identifies each constructed region and the Boolean operation that is to be used to create it. Boolean operations will be performed in the order in which they appear in the region definition section. As seen in the example of Figure 4-10, a space or blank character may be used as an optional operator. If the operator is blank and the following solid/region number is positive then the operator is assumed to be a "union". Otherwise, if the solid/region number is negative then the operator is assumed the "difference"

The region identification section simply contains a list of all region IDs and their associated attributes. Figure 4-10 contains a sample of a complete geometry description file.

There is one special line of data that is required in the description file that is not part of the geometry description or construction. Between the region definition section and region identification section, the ray tracer expects to find the value of -1 in columns 1-5; this is used as a delimiter to mark the end of the region definition section.

TITLE record	
Columns	*Contents*
1-5	Model units (in, ft, mm, cm, m)
6-65	Name for targets

CONTROL record	
Columns	*Contents*
1-5	Number of primitives
6-10	Number of regions

PRIMITIVE DEFINITION records
See Table A

REGION DEFINITION records	
Columns	*Contents*
1-5	Region number
7-8	Boolean operator
9-13	Primitive number
14-15	Boolean operator
16-20	Primitive number
21-22	Boolean operator
23-27	Primitive number
28-29	Boolean operator

30-34	Primitive number
35-36	Boolean operator
37-41	Primitive number
42-43	Boolean operator
44-48	Primitive number
49-50	Boolean operator
51-55	Primitive number
56-57	Boolean operator
58-62	Primitive number
63-64	Boolean operator
65-69	Primitive number
71-80	Comments

Notes on Boolean operations:
"DIFFERENCE" – A negative primitive number
"INTERSECTION " – A positve primitive number
"UNION" – 'or' in the Boolean operator column
The union operation is performed between the two sets of primitives that are listed before and after an 'or" operator . e.g., 2 2 -4 or 5 6 or 7 -1
(region) (solids…)

Their operations for this example will be performed in the following order:

1) DIFFERENCE between primitives 2 and 4
2) INTERSECTION between primitives 5 and 6
3) UNION the results of steps 1 and 2
4) DIFFERENCE between primitives 7 and 1
5) UNION the results of steps 3 and 4

In this example, implied parentheses exist around (2 -4), (5 6) and (7 -1).
Other examples of operations include:
2 2 -4 6 or 7 -1
In this example, the DIFFERENCE is taken between primitives 2 and 4, and then the INTERSECTION is taken between that result and primitive 6. A DIFFERENCE is taken between primitives 7 and 1. The UNION of these results is then taken.

REGION IDENTIFICATION record	
Columns	*Contents*
1-3	Region number
4-10	Component code number (1-9999)
11-15	Space code number (1-99)
16-20	Material code
21-30	(unused)
31-80	Region description

Geometry Description Record Formats	
Half Space	HAF
Arbitrary Tetrahedron	ARB4
Polyhedron - 5 Vertices	ARB5
Polyhedron - 6 Vertices	ARB6
Arbitrary Wedge	ARW
Right Angle Wedge	RAW
Polyhedron - 7 Vertices	ARB7
Polyhedron - 8 Vertices	ARB8
Box	BOX
Rectangular Parallelepiped RPP	
Triangular - faceted Polyhedron	ARS
Truncated General Cone	TGC
Truncated Elliptical Cone	TEC
Truncated Right Cone	TRC
Right Elliptical Cylinder	REC
Right Circular Cylinder	RCC
Right Parabolic Cylinder	RPC
Right Hyperbolic Cylinder	RHC
Elliptical Paraboloid	EPA
Elliptical Hyperboloid	EHY
General Ellipsoid	ELLG
Ellipsoid of Revolution	ELL
Sphere	SPH
Elliptical Torus	ETO
Circular Torus	TOR

Cols 1-5	6-8	9-10	11-20	21-30	31-40	41-50	51-60	61-70	71-80
No.	RPP		xmin	xmax	ymin	ymax	zmin	zmax	comments
No.	BOX		V_x	V_y	V_z	H_x	H_y	H_z	comments
No.			W_x	W_y	W_z	D_x	D_y	D_z	comments
No.	RAW		V_x	V_y	V_z	D_x	D_y	D_z	comments
No.			H_x	H_y	H_z	W_x	W_y	W_z	comments
No.	SPH		V_x	V_y	V_z	R			comments
No.	ELL		V_x	V_y	V_z	A_x	A_y	A_z	comments
No.			R						comments
No.	TOR		V_x	V_y	V_z	N_x	N_y	N_z	comments
No.			R_1	R_2					comments
No.	RCC		V_x	V_y	V_z	H_x	H_y	H_z	comments
No.			R						comments
No.	REC		V_x	V_y	V_z	H_x	H_y	H_z	comments
No.			A_x	A_y	A_z	B_x	B_y	B_z	comments
No.	TRC		V_x	V_y	V_z	H_x	H_y	H_z	comments
No.			R_1	R_2					comments
No.	EHY		V_x	V_y	V_z	H_x	H_y	B_z	comments
No.			A_x	A_y	A_z	R_1	R_2	C	comments
No.	EPA		V_x	V_y	V_z	H_x	H_y	H_z	comments
No.			A_x	A_y	A_z	R_1	R_2		comments
No.	RHC		V_x	V_y	V_z	H_x	H_y	H_z	comments
No.			B_x	B_y	B_z	R	C		comments
No.	ARW		V_x	V_y	V_z	$H1_x$	$H1_y$	$H1_z$	comments
No.			$H2_x$	$H2_y$	$H2_z$	B_x	B_y	B_z	comments
No.	RPC		V_x	V_y	V_z	H_x	H_y	H_z	comments
No.			B_x	B_y	B_z	R			comments
No.	ARB4		X_1	Y_1	Z_1	X_2	Y_2	Z_2	comments
No.			X_3	Y_3	Z_3	X_4	Y_4	Z_4	comments
No.	ARB5		X_1	Y_1	Z_1	X_2	Y_2	Z_2	comments
No.			X_3	Y_3	Z_3	X_4	Y_4	Z_4	comments
No.			X_5	Y_5	Z_5				comments

Figure 4-8. Solid Primitive Definitions

Cols 1-5	6-8	9-10	11-20	21-30	31-40	41-50	51-60	61-70	71-80
No.	ARB6		X_1	Y_1	Z_1	X_2	Y_2	Z_2	comments
No.			X_3	Y_3	Z_3	X_4	Y_4	Z_4	comments
No.			X_5	Y_5	Z_5	X_6	Y_6	Z_6	comments
No.	ARB7		X_1	Y_1	Z_1	X_2	Y_2	Z_2	comments
No.			X_3	Y_3	Z_3	X_4	Y_4	Z_4	comments
No.			X_5	Y_5	Z_5	X_6	Y_6	Z_6	comments
No.			X_7	Y_7	Z_7				comments
No.	ARB8		X_1	Y_1	Z_1	X_2	Y_2	Z_2	comments
No.			X_3	Y_3	Z_3	X_4	Y_4	Z_4	comments
No.			X_5	Y_5	Z_5	X_6	Y_6	Z_6	comments
No.			X_7	Y_7	Z_7	X_8	Y_8	Z_8	comments
No.	TEC		V_x	V_y	V_z	H_x	H_y	H_z	comments
No.			A_x	A_y	A_z	B_x	B_y	B_z	comments
No.			ratio						comments
No.	TGC		V_x	V_y	V_z	H_x	H_y	H_z	comments
No.			A_x	A_y	A_z	B_x	B_y	B_z	comments
No.			T(A)	T(B)					comments
No.	HAF		N_x	N_y	N_z	D_h			comments
No.	ETO		V_x	V_y	V_z	N_x	N_y	N_z	comments
No.			C_x	C_y	C_z	R	R_d		comments
No.	ELLG		V_x	V_y	V_z	X_x	X_y	X_z	comments
No.			Y_x	Y_y	Y_z	Z_x	Z_y	Z_z	comments
No.	ARS		M	N					comments
No.			$X_{1,1}$	$Y_{1,1}$	$Z_{1,1}$	$X_{1,2}$	$Y_{1,2}$	$Z_{1,2}$	comments
.			.						.
.			.						.
.			.						.
No.			$X_{1,N}$	$Y_{1,N}$	$Z_{1,N}$				
No.			$X_{2,1}$	...					
.			.						
.			.						
No.			$X_{M,1}$	...					
.			.						
.			.						
No.			$X_{M,N}$	$Y_{M,N}$	$Z_{M,N}$				

Figure 4-9. Solid Primitive Definitions (continued)

For ARS the symbol 'M' represents the number of curves and the symbol 'N' represents the number of points required to define the curve with the most number of points. (eg. If M = 2 where curve 1 requires 3 points to define it and curve 2 requires 4 points then the value of N must be 4.)

The Equation for a half plane is: NxX + NyY + NzZ = Dh

Where V is the vector, D is the depth vector, H is the height vector, W is the width vector, N is the normal vector, R is the radius, A is the semi-major axis of ellipse, and B is the semi-minor axis of ellipse.

An example file follows.

```
m   Simple test object in meter units
 2  2
 1rpp   -1.5000  1.5000  -1.5000  1.5000  -1.5000  1.5000
 2rpp   -0.5000  0.5000  -0.5000  0.5000  -0.5000  0.5000
 1    1   -2
 2    2
-1
 1  100   0   0   0    main box
 2  100   2   0   0    inside box
```

Figure 4-10: Example.cg file of a hollow box

4.2.2.1.2 Image Formation Inputs

4.2.2.1.2.1 Input Variables

The input variables for the image formation benchmark will be stored as ASCII text. Each item will be on a separate line, with the variable number first followed by a text comment identifying that line of the input variable file. The system IPR will be written out in the text file in row column order. There will be a separate Input Variable file for each input data set.

Input data file name	This will be the name of the Input data file that will be used with this Input Variable file.
Aperture specification	The aperture specification here will include the number of samples in range and cross range and image resolution in meters.
System IPR	This will be an array of floats representing the convolution kernel. The kernel can be as small as 3 x 3, where only the main lobe information will appear in the final image, or as large as 35 x 35, where numerous side lobes appear in the final image.

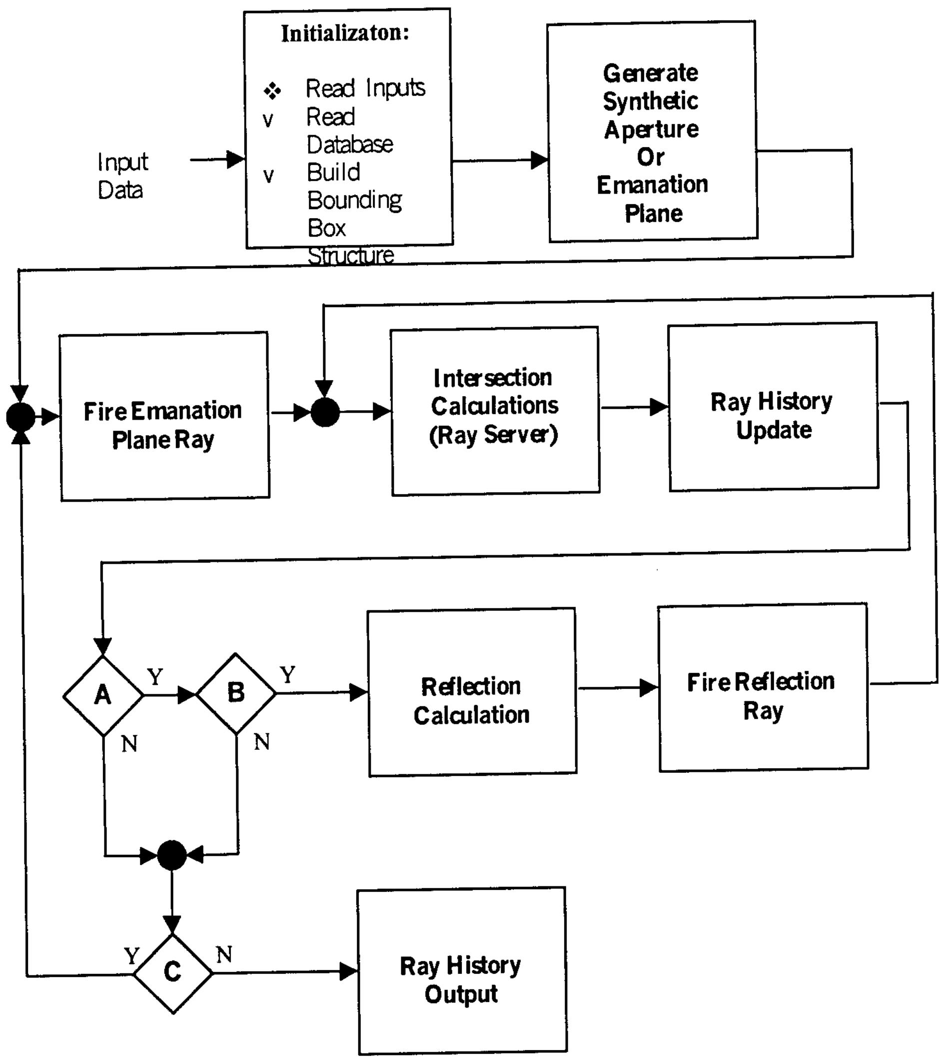
4.2.2.1.2.2 Input Data

The input data arrays are EM responses from the ray history arrays. The EM responses are stored in complex format. The ray history and EM responses will come in three sizes and in both float and double precision formats. These are ray histories with EM responses added right after the subpart name.

- A small array at 512 X 512 sample points,
- A medium array at 2048 X 2048 sample points, and
- A large array at 4096 X 4096 sample points.

4.2.2.2 Algorithmic Specification

4.2.2.2.1 Recursive Ray Tracing Benchmark Algorithm Specifications



A? Did the last ray have and intersection
(Yes - continue No - The rays has left the database)

B? Is the reflection Count < Max # of Reflections
(Yes - Continue No - End of this ray)

C? Are they any more rays to be fired from the Emanation Plane
(Yes - Fire Next Ray No - Output Ray History

4.2.2.2.1.1 Initialization

The initialization process reads in the input variables and input database. It then builds the bounding-box structure and object structure from the input database. The bounding-box structure contains the bounding hierarchy and each bounding surface contains tags to the object structure. In the initialization the Emanation Ray History pointer array is created and initialized.

Both the bounding-box structure and the object structure should be available to all routines.

Because the BRL-CAD.cg format does not support bounding boxes, a separate file will be provided that contains the bounding information. The format of this bounding-box file is the same as that of the polygon models, except it will contain no polygon data. The object and sub-object name will match the names found in the .cg file thus providing the linkage between the objects and their bounding boxes.

4.2.2.2.1.2 Generation of the Emanation Plane

Using the aperture specification and the number of sample points, from the input data, an Emanation plane is generated.

If the FOV is ZERO, and it always will be for this benchmark, this means that a parallel projection is to be used. With this projection, all the rays are fired in the look direction. The only thing that needs to be determined is the position and scale of the sample points. For this benchmark, we will assume that for the parallel projection case the target will be centered in the sample space and that the top-level bounding box just fits into the sample window. This means that the target translation is ignored and only the rotation is used. Each point in the top level bounding box must be put through the target rotation matrix and then through the viewing matrix. The viewing matrix is built from the rotations needed to rotate the target into the viewing azimuth and elevation. This rotation can be derived using the following pseudo code.

```
Assume that z is up and create a vector up = [0,0,1].  Find the vector
that is the cross product of the lookat vector and the up vector.
Make sure that this new vector is a unit vector and call it alpha
Now find the cross product of the lookat vector and the alpha vector.
Again, make sure it is a unit vector and call it Beta.
The 3x3 Euler rotation matrix is then:
      Alpha_i,     Alpha_j,     Alpha_k
      Beta_i,      Beta_j,      Beta_k
      Lookat_i,    Lookat_j,    Lookat_k
```

Keep this viewing rotation matrix around, as it will be used in the generation of the emanation rays in the next step. It is possible at this point to combine the target rotation matrix and the viewing matrix, so only one matrix multiplication has to be done.

When these rotations are applied to the top-level bounding-box points, a new bounding box is derived that shows the target extent in the viewing frame, if the maximum distance in the horizontal and vertical directions are taken. The Delta size of the viewing window or Emanation plane is known in target units. To get the needed size of an individual pixel, take the larger of the two and divide by the number of sample points in that direction. This benchmark will assume square pixels; thus, a pixel will have the same dimension in both the vertical and horizontal. This same bounding information can be used to center the target in the sample space.

The result of this will be a (x, y) coordinate in viewing space based on the row and column in the sample space.

The distance from the radar to the center of the target is found by subtracting the target translation vector from the radar location vector. The magnitude of this difference vector is the desire distance number.

4.2.2.2.1.3 Emanation Plane Ray Firing

This routine is the beginning of the outside loop the Ray tracing system. It takes the information derived in generating the Emanation plane and produces global starting coordinates and direction vectors for the out going rays. Every time it is called a new starting ray position and direction vector are calculated for the next sample point in the Emanation plane.

It generates the starting point in global coordinates by taking the row and column numbers and using the scaling information from the generation routine to derive an x, y, z point in viewing space. The distance from the center of the target to the radar is added to the z value to arrive at the viewing coordinate. This is then converted into a global coordinate by passing this vector through the inverse of the viewing and target rotation matrices. This global coordinate is then the starting position for the current ray.

The direction vector for this benchmark, because we are only using parallel projections, is just the look direction vector from the input variables, aperture specification.

A flag is set that lets the ray history update routine know that a new emanation plane rays has been fired. The starting position and direction are then passed to the Ray Server.

This routine is called on the first ray, when the previous ray leaves the target database, or when the maximum number of reflections, of the previous ray, has been reached.

4.2.2.2.1.4 Intersection Calculations (Ray Server)

The ray server receives a ray, starting position and direction vector, and determines what, if any, objects in the target database this ray intersects. Once the intersections are found, they are processed and the nearest surface intersection is returned. The Ray Server is divided into two major parts.

4.2.2.2.1.5 Bounding-Box Intersections

The first section is the bounding box intersection process. Here the bounding-box hierarchy is tested against the ray. The top level bounding box is tested first. If it is intersected then each object bounding box is tested. If the current ray does not intersect the top-level box, the ray server returns a flag that no intersections were found. This process continues down the bounding box hierarchy until all possible intersected objects have been found.

4.2.2.2.1.6 Object Intersection

The second section finds the actual points of intersection, if they exist, for each object identified in the above process. Once the intersection points have been identified they are sorted by distance from the ray starting point and the nearest intersection is returned. The ray server also returns the surface normal information at the intersection point.

Polygon Models	With Polygon models, this is a straightforward process of testing each polygon, in each sub-object, against the ray under test. If an intersection is found it is recorded in the intersection list array. When all the intersections are found, the intersection nearest the starting point of the ray under test is return along with the surface normal at that polygon. In this benchmark, all the polygons are three sided and thus the surface normal is easily calculated using the polygon vertices. It should be noted that the normal is the same for any point on that polygon.
Solid Models with CSG	With solid models and CSG operators, things are a little more difficult. Finding the intersections with the solid models requires more effort, due to their more complex shapes. The intersections come in pairs. One intersection enters the object and the other leaves the object. The real work comes in using the intersection pairs and the CSG operators to find the true intersection point for any object. The supported CSG operators can be seen in Figure 4-11. If a ray were to pass through center top of the cube in Figure 4-11, the intersection pairs would be operated on as shown in Figure 4-12. The CSG operations are executed as written in the .cg file. It is noted here, that with solids, the bounding boxes for objects surround a complete CSG object. The sub-object bounding boxes surround the basic solid primitives that are combined, using the CSG operators, to form the complete CSG object. In the example in Figure 4-11, the object bounding box would surround the cube and cylinder. The sub-object bounding boxes would be around the cube as sub-object 1 and cylinder as sub-object 2.

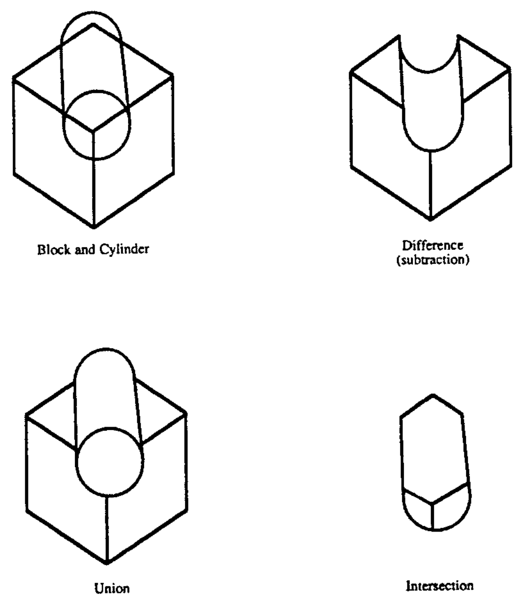


Figure 4-11: Supported CSG Operators

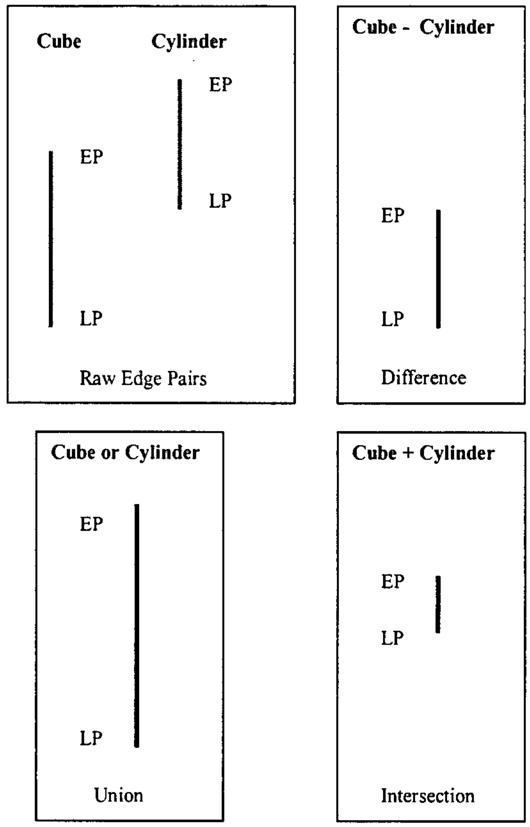


Figure 4-12: CSG Operators on Intersection Pairs

4.2.2.2.1.7 Surface Intersection Algorithms

All the surface intersection algorithms needed for this benchmark can be found in " An Introduction to Ray Tracing", Edited by Andrew S. Glassner, Academic Press, ISBN 0-12-286160-4. This is referenced in the Bibliography section, but it is given special note here. This is a single resource that has much, if not all, the needed information about Ray Tracing algorithms. It is felt that the coverage of intersection algorithms on pages 33 through 119 of that text is much better and less confusing than what this author could put forth.

4.2.2.2.1.8 Ray History Update

This routine handles the ray history linked list. There are three possible functions that may run.

If the ray that is being processed is a new ray from the emanation plane, the ray history update routine starts a new ray history link and generates a new node structure, initializes the node, records the information returned by the ray server, and adds one to the total intersection count.

If the ray that is being processed is a reflected ray, the ray history update generates a new node structure, initializes it, and places the appropriate links in the new node, the previous node, and records the information returned by the Ray Server. It also updates the previous node with the reflected ray information that is now available. It then adds one to the total intersection count.

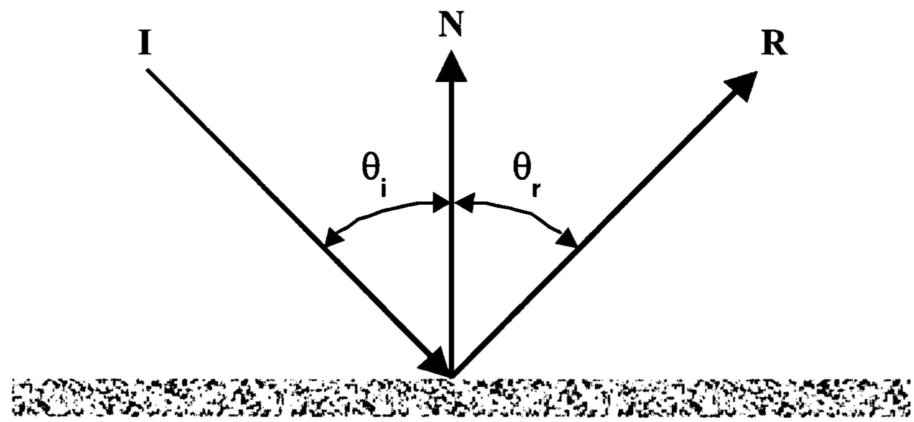
If the ray that is being processed returns from the Ray Server without finding an intersection, the ray history server updates the current node with the reflected ray information and then does nothing else.

The node generation consists of allocating a new section of memory for the new node structure and then initializes it by entering a NULL into the previous_node and next_node pointer variables.

The process of starting a new ray history link involves placing a pointer to the newly generated node structure into the emanation ray history pointer array. This array has the same dimensions as the emanation plane and array keeps a pointer to each ray history node associated with a new ray being fired from the emanation plane.

4.2.2.2.1.9 Reflection Calculations

A reflection ray is calculated for every intersection point as long as the maximum number of intersections has not been exceeded. The process of finding a reflection ray is base on Snells' law for a perfect specular reflection. This law states that an incoming ray of will reflect at an equal and opposite angle, relative to the normal at the reflection point:



$$\theta_i = \theta_r$$

The formula for the direction of a specularly reflected ray is

$$\mathbf{R} = \mathbf{I} - 2(\mathbf{N} \bullet \mathbf{I})\mathbf{N}$$

where **I** is the incident ray, **N** is the normal at the reflection point, and **R** is the reflected ray.

4.2.2.2.1.10 Reflection Ray Firing

The reflection ray firing is similar to the Emanation ray firing. In this case, the reflected ray calculated above becomes the look direction vector and the intersection point becomes the starting position of the ray. These are passed to the Ray Server.

4.2.2.2.1.11 Ray History Output

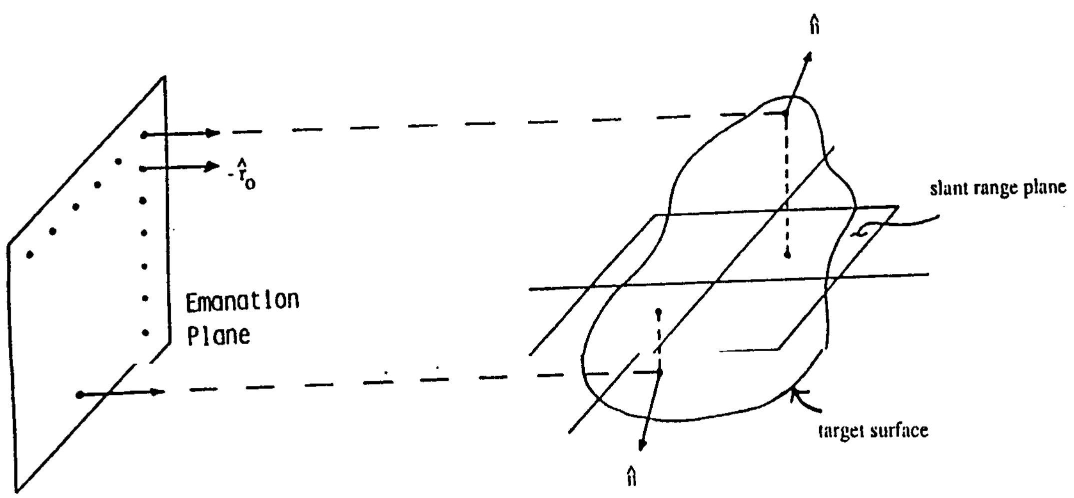
After all the emanation rays have been fired and their reflection paths followed, the final step in the geometry sampling process is to output the ray history. This is done by using the emanation ray history pointer array to follow each ray history. As each node is traversed the data in the node structure is written out to the ray history file in the form specified in section 4.2.2.3.1. As this data is written out, the pointers for previous_node and next_node get replace with index numbers as if each node was an entry in a large 1-D array of nodes. Those nodes that have NULL pointers are replaced by -1.

4.2.2.2.2 Image Formation Benchmark Algorithm Specifications

The Image Formation process consists of three steps that convert the EM Contributions into a slant plane SAR Image.

4.2.2.2.2.1 Mapping EM contributions

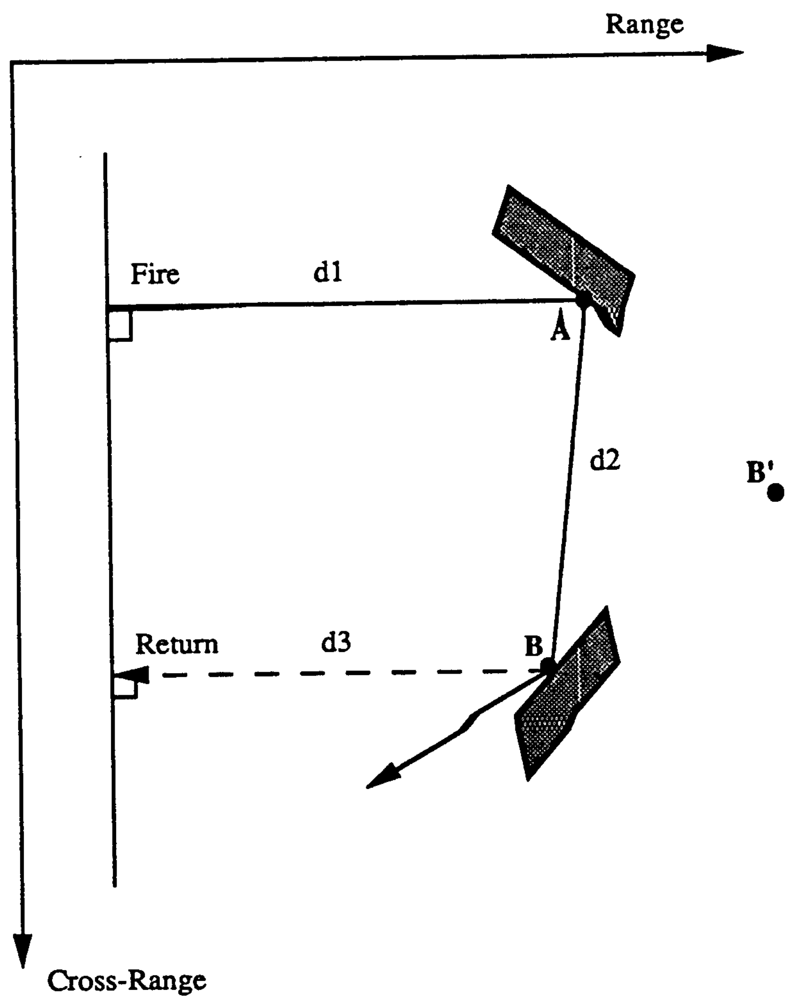
The first step in the image formation process is to map the EM contributions of each ray history into the slant plane. The slant plane image is a grid of cells measured in range and cross range (see Figure 4-13).



- sequentially cast rays parallel to $\hat{r}_o$ to generate ray history of target sample points
- compute integrated reflectivity for each sample point
- compute slant range coordinates for unique contribution
- weight reflectivity by spatial impulse response and sum into slant range plane (input centered convolution)

Figure 4-13: Formation of the SAR Image

The first intersection EM contribution is mapped into this plane based on the position in the emanation plane, for the cross range position, and the distance from the firing point to the intersection point, for the range position. The multiple reflections are mapped to a mean aspect and more distant range consistent with their appearance in the SAR imagery (See Figure 4-14).



For Reflection A:

Contribution is mapped directly at A in SAR image

For Reflection B:

Contribution is mapped at B' in SAR image

Range equals total distance (d1+d2+d3) divided by 2
Cross-range position is average of cross-range positions of A and B.

Figure 4-14: SAR Mapping of Returns

This process is followed for each ray history and each EM contribution for each reflection resulting in a complex data array.

4.2.2.2.2.2 IPR Convolution

Once the EM scattering is mapped to the slant plane, it must be convolved with the system IPR. This is an input centered convolution. The convolution edge effects are accounted for by mirroring the edge pixels of the slant plane. Pixels are copied from the left and right edges, range, and then copied from the top and bottom, cross-range. This insures the corners of the mirrored images are filled correctly. The number of pixels copied is equal to half of the IPR convolution width. This is a standard 2-D convolution process.

4.2.2.2.2.3 Detection

The final process in the Image Formation sequence is the Detection process. This is nothing more than finding the Magnitude image of the complex image plane. The output from this process forms a viewable, or real-valued, image and is the desired output for this portion of the benchmark.

4.2.2.3 Output

4.2.2.3.1 Recursive Ray Tracing Benchmark Output

The output for the Recursive Ray Tracing Benchmark is a link list of the intersection information for each sample in the aperture. This ray history file will be a binary file based on an array of the following structure with the first number in the file being the number of array entries

```
int  number_of_entries
struct  Ray_History {
      char  object_name[256];
      char  part_name[256];
      char  subpart_name[256];
      float       intersection_x;
      float       intersection_y;
      float       intersection_z;
      float       normal_vec_i;
      float       normal_vec_j;
      float       normal_vec_k;
      float ray_length;
      float ray_start_i;
      float       ray_start_j;
      float       ray_start_k;
      float       ray_vector_i;
      float       ray_vector_j;
      float       ray_vector_k;
      float       reflection_vector_i;
      float       reflection _vector_j;
      float       reflection _vector_k;
      int   previous_node;
      int   next_node;
}
```

If the previous_node value is -1 this is the beginning of a ray history. If the next_node has a value of -1 this is the last intersection point for this ray history. Using the first intersection point and the normal of each ray history one can create a shaded image of the target under test. This can be a good tool for evaluating the ray tracer.

4.2.2.3.2 Image Formation Benchmark Output

The outputs for the Image Formation Benchmark will be, float or double, binary files output at each stage of the process. The final binary file, after detection, is the only file that is not complex. When displaying these images one should us a dB scaling do to the nature of the detected image. Using -40dB down from the peak value will provide a good-looking image. Other conversions that don't account for the wide bandwidth in the final image may hide defects. When doing a timing run, only the final detected or real valued image should be output. The in-between images are used only to validate each step in the process.

4.2.2.4 Acceptance Test

A given data set will be considered successfully executed when the processing sequence results match with the corresponding output provided with the benchmark. Precision is discussed in Section 3.7.

4.2.2.4.1 Acceptance Test for Recursive Ray Tracer Benchmark

It must be realized that small differences in intersection location and in the calculation of the normal can result in large errors after several reflections. Acceptance of this section of the benchmark should look at each level in the ray history to see if it meets the tolerances discussed in Section 3.7.

4.2.2.4.2 Acceptance Test for the Image Formation Benchmark

An output data set for each step in the process is provided and a match should be achieved for each step in the process.

4.2.2.5 Metrics

4.2.2.5.1 Metrics for Recursive Ray Tracer Benchmark

The primary metric for the Recursive Ray Tracer Benchmark is the total time to complete the evaluation of all rays in the given aperture. Secondary metrics are based on processor loading as a function of time. Also where possible a metric should be attempted that gives the scalability ratios for input database size, aperture resolution, and number of processors. These secondary metrics will provide useful information on known problems with parallel ray-tracing applications.

4.2.2.5.2 Metrics for the Image Formation Benchmark

The primary metric for the Image Formation benchmark is total time to complete all the steps with the given input data set. A secondary metrics consists of the individual times for each step in the Image Formation process.

4.2.2.6 Baseline Source Code

Baseline source code is available at http://www.aaec.com/projectweb/dis.

4.2.2.7 Baseline Performance Figures

Baseline performance figures are available at http://www.aaec.com/projectweb/dis.

4.2.2.8 Test Data Sets

Test data sets are available at http://www.aaec.com/projectweb/dis.

4.2.2.9 References

Ray Tracing References

[1] K. Bouatouch and T. Priol. Parallel space tracing: An experience on an iPSC hypercube. In N. Magnenat-Thalmann and D. Thalmann, editors, *New Trends in Computer Graphics (Proceedings of CG International '88)*, pages 170–187, New York, 1988. Springer-Verlag.

[2] J. G. Cleary, B. M. Wyvill, G. M. Birtwistle, and R. Vatti. Multiprocessor ray tracing. *Computer Graphics Forum*, pages 3–12, 1986.

[3] F. C. Crow, G. Demos, J. Hardy, J. McLaugglin, and K. Sims. 3d image synthesis on the connection machine. In *Proceedings Parallel Processing for Computer Vision and Display*, Leeds, 1988.

[4] M. A. Z. Dipp′ e and J. Swensen. An adaptive subdivision algorithm and parallel architecture for realistic image synthesis. *ACM Computer Graphics*, 18(3):149–158, Jul 1984.

[5] S. A. Green and D. J. Paddon. Exploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics and Applications*, pages 12–27, Nov 1989.

[6] H. Kobayashi, S. Nishimura, H. Kubota, T. Nakamura, and Y. Shigei. Load balancing strategies for a parallel ray-tracing system based on constant subdivision. *The Visual Computer*, 4(4):197–209, 1988.

[7] A. J. F. Kok. *Ray Tracing and Radiosity Methods for Photorealistic Image Synthesis*. PhD thesis, Delft University of Technology, jan 1994.

[8] T. T. Y. Lin and M. Slater. Stochastic ray tracing using SIMD processor arrays. *The Visual Computer*, 7:187–199, 1991.

[9] D. J. Plunkett and M. J. Bailey. The vectorization of a ray-tracing algorithm for improved execution speed. *IEEE Computer Graphics and Applications*, 5(8):52–60, aug 1985.

[10] T. Priol and K. Bouatouch. Static load balancing for a parallel ray tracing on a MIMD hypercube. *The Visual Computer*, 5:109–119, 1989.

[11] E. Reinhard. Hybrid scheduling for parallel ray tracing. TWAIO final report, Delft University of Technology, jan 1996.

[12] I. D. Scherson and C. Caspary. A self-balanced parallel ray-tracing algorithm. In P. M. Dew, R. A. Earnshaw, and T. R. Heywood, editors, *Parallel Processing for Computer Vision and Display*, volume 4, pages 188–196, Wokingham, 1988. Addison-Wesley Publishing Company.

[13] L. S. Shen, E. Deprettere, and P. Dewilde. A new space partition technique to support a highly pipelined parallel architecture for the radiosity method. In *Advances in Graphics Hardware V, proceedings Fifth Eurographics Workshop on Hardware*. Springer-Verlag, 1990.

[14] E. R. Frederik, W. Jansen . Rendering Large Scenes Using Parallel Ray Tracing. *Parallel Computing*, pages 873-885, 1997

[15] T. Wilson, N. Doe. Acceleration Schemes for Ray Tracing. Report Number: CS-TR-92-22, Department of Computer Science, University of Central Florida, September 1992.

[16] R.L. Cook, T. Porter, L. Carpenter. Distributed Ray Tracing. *Computer Graphics* (Proceedings of SIGGRAPH 1984), 18(3), 137-145, July 1984.

[17] R.L. Cook. Stochastic sampling in computer graphics, ACM Transaction in Graphics 5(1), 51-72, January 1986.

[18] A. S. Glassner (Editor), *An Introduction to Ray Tracing*, Academic Press 1989.

[19] Ray Tracing Bibliography, http://www.cm.cf.ac.uk/Ray.Tracing/RT.Bibliography.html

Simulated SAR References

[1] D.J. Andersh, M. Hazlett, S.W. Lee, D.D. Reeves, D.P. Sullivan and Y. Chu, "Xpatch: A high frequency electromagnetic-scattering prediction code and environment for complex three-dimensional objects," *IEEE Antennas & Propagation. Magazine,* vol. 36, pp.65-69, 1994.

[2] J. Baldauf, S.W. Lee, L. Lin, S.K. Jeng, S.M. Scarborough, and C.L. Yu, "High frequency scattering from trihedral corner reflectors and other benchmark targets: SBR vs. experiment," *IEEE Transacrions on Antennas and Propagation,* vol. 39, pp. 1345-1351, 1991.

[3] R. Bhalla and H. Ling, *Image-domain ray tube integration formula for the shooting and bouncing ray technique,* University of Texas Report, NASA Grant NCC 3-273, July 1993.

[4] R. Bhalla and H. Ling, "A fast algorithm for signature prediction and image formation using the shooting and bouncing ray technique," to appear in *IEEE Transactions on Antennas and Propagation,* 1995.

[5] G. Franceschetti, M. Migliaccio, D. Riccio, and G. Schirinzi, "SARAS: A Synthetic Aperture Radar (SAR) Raw Signal Simulator," *IEEE Transactions on Geoscience and Remote Sensing,* Vol. 30, No. 1, January 1992.

[6] G. Franceschetti, M. Migliaccio, and D. Riccio, "SAR Raw Signal Simulation of Actual Ground Sites in Terms of Sparse Input Data," *IEEE Transactions on Geoscience and Remote Sensing,* Vol. 32, No. 6, November 1994.

[7] D.E Herrick and I.J. LaHaie, *SRIM Polarimetric Signature Modeling,* ERIM IR&D Final Report 675805-1-F, December 1988.

[8] D.E Herrick and B.J. Thelen, "Computer Simulation of Clutter in SAR Imagery," *Proceedings of the Progress in Electromagnetics Research Symposium,* Cambridge, MA, July 1991

[9] D.E Herrick, "Computer Simulation of Polarimetric Radar and Laser Imagery," in *Direct and Inverse Methods in Radar Polarimetry,* W.-M. Boerner *et al.* (eds), Klumer Academic Publishers, The Netherlands 1992.

[10] D.E Herrick, M.A. Ricoy, and W.D. Williams, "Modeling of Foliage Effects in UHF SAR", *Proceedings qfthe Ground Target Modeling and Validation Conference,* Houghton, MI, August 1994.

[11] D.E Herrick, M.A. Ricoy, and W.D. Williams, "Synthesizing SAR Signatures of Ground Vehicles with Complex Scattering Mechanisms", *Proceedings of the Ground Target Modeling and Validation Conference,* Houghton, MI, August 1994.

[12] E.R. Keydel, D.E *Henick,* and W.D. Williams, "Interactive Countermeasures Design and Analysis Tool," *Proceedings of the Ground Target Modeling and Validation Conference,* Houghton, MI, August 1994.

[13] S.W. Lee and D.J. Andersh, *On Nussbaum Method for Exponential Series,* Electromagnetic Laboratory Technical Report ARTI-92-11, University of Illinois, Urbana, November, 1992.

[14] H. Ling, R.C. Chou, and S.W. Lee, "Shooting and Bouncing Rays: Calculating the RCS of an arbitrarily shaped cavity," *IEEE Transactions on Antennas and Propagation,* Vol. 37, pp. 194-05, 1989.

[15] J.M. Nasr and D. Vidal-Madjar, "Image Simulation of Geometric Targets for Spaceborne Synthetic Aperture Radar", *IEEE Transactions on Geoscience and Remote Sensing,* Vol. 29, No. 6, November 1991.

[16] N.D. Taket, S.M. Howarth, and R.E. Burge, "A Model For the Imaging of Urban Areas by Synthetis Aperture Radar," *IEEE Transactions on Geoscience and Remote Sensing,* Vol. 29, No. 3, May 1991.

[17] M.R. Wohlers, S.Hsiao, J. Mendelsohn, and G. Gerdner, "Computer Simulation of Synthetic Aperture Radar Images of Three-Dimensional Objects," *IEEE Transactions on Aerospace and Electronic Systems,* Vol. AES-16, No. 3, May 1980.

4.2.3 Image Understanding

Algorithms were selected for this benchmark that perform spatial filtering to determine regions of interest (ROIs) and operate on a set of ROIs. The *Image Understanding* benchmark consists of a sequence of components depicted in Figure 4-15. Also included in the figure are names for input parameters, images, and intermediate output at different parts of the sequence, which are referred to later in this document. The morphological filter component provides a spatial filter to remove background clutter in the image. Next, the ROI selection component applies a threshold to determine target pixels, groups these pixels into ROIs, and selects a subset of ROIs depending on specific selection logic. Finally, the feature extraction component operates over and computes features for the selected ROIs.

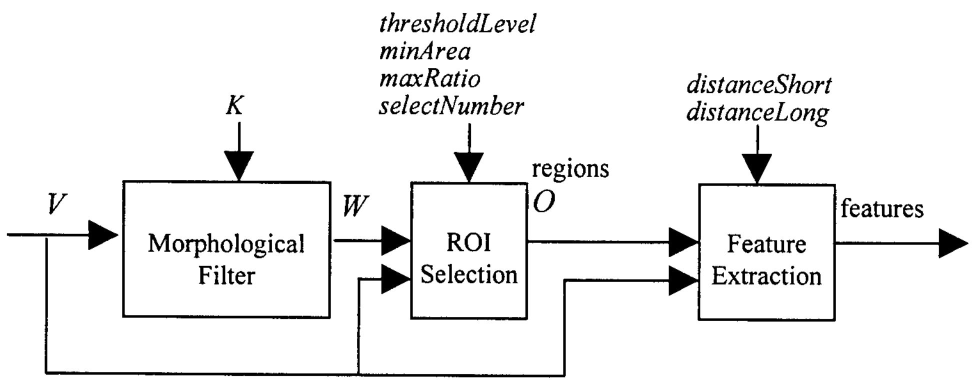


Figure 4-15: Image Understanding Sequence

The input required by the sequence is a set of parameters and an image, *V*. The first step in the sequence is a spatial morphological filter component generating image *W*. Then, the ROI selection component performs a thresholding and groups connected pixels into ROIs (or targets) contained in image *W*. This component then computes initial features for each ROI in image *W*, and selects a list of ROIs depending on the values of these features. These selected ROIs are stored in object image, *O*. The initial features for each selected ROI are stored in list, *regions*. Lastly, the feature extraction component computes additional features for the selected ROIs. The output at the end of the sequence is a feature list, *features,* with both sets of features computed for each selected ROI.

4.2.3.1 Input

An input data set for the Image Understanding benchmark, which contains all input required for a single run of the benchmark, is provided in one binary input file, in the following order:

input image *V* (stored as an array of short integers)

morphological kernel *K* (stored as an array of unsigned bytes)

thresholdLevel (short integer)

minimum acceptable area value, *minArea* (integer)

maximum acceptable perimeter-to-area ratio, *maxRatio* (float) number of ROIs to select, *selectNumber* (integer), and

distanceShort and *distanceLong* (integers).

The above values are provided in binary representation for an 8-bit unsigned byte, 2-byte short integer, 4-byte integer, or 4-byte float as described in Section 3.6. References to these inputs are found in Figure 4-15 and in the sections describing each component of the sequence below. Descriptions or formats for input image V and kernel K are provided in the following subsections.

4.2.3.1.1 Image V

Images provided as input for the benchmark are rectangular, with square pixels, and stored in row-dominant order. Let image V have X columns and Y rows, and v(x,y) be any pixel in V where $0 \le x < X$ and $0 \le y < Y$, as shown here:

v(0,0)	v(1,0)	v(2,0)	• • •	v(X-1,0)
v(0,1)	v(1,1)	v(2,1)	• • •	v(X-1,1)
v(0,2)	v(1,2)	v(2,2)	• • •	v(X-1,2)
• • •	• • •	• • •		• • •
v(0,Y-1)	v(1,Y-1)	v(2,Y-1)	• • •	v(X-1,Y-1)

Figure 4-16: Sample Image V with X columns and Y rows

Preceding the image data are two *integers*, representing the number of columns and the number of rows, respectively. Next, pixel values are provided, as *short integers*, in row-dominant order, as shown in the following table. Therefore, an image with three rows and five columns requires thirty-eight bytes, where the first eight bytes contain two 4-byte *integer* values indicating the number of columns and rows, followed by fifteen 2-byte *short integer* pixel values.

Table 4.2.3-1: File containing byte image V

byte offset ↓	contents	byte offset ↓	contents	byte offset ↓	contents
0		**6+2*X***	*v(X-1,0)*	•	•
2	*X*	**8+2*X***	*v(0,1)*	•	•
4		•	•	**6+2*XY*-2*X***	*v(X-1,Y-2)*
6	*Y*	•	•	**8+2*XY*-2*X***	*v(0,Y-1)*
8	*v(0,0)*	**6+4*X***	*v(X-1,1)*	•	•
10	*v(1,0)*	**8+4*X***	*v(0,2)*	•	•
•	•	•	•	**4+2*XY***	*v(X-2,Y-1)*
•	•	•	•	**6+2*XY***	*v(X-1,Y-1)*

4.2.3.1.2 Kernels

Kernels are small images that define a neighborhood or window to be used in the processing of a larger image. Kernels are provided in the same format as images and an example of the file containing a kernel is contained in Table 4.2.3-2 below. Note that a kernel with three rows and five columns requires twenty-three bytes, where the first eight bytes contain two 4-byte *integer* values indicating the number of columns and rows, followed by 15 *unsigned-byte* pixel values.

Table 4.2.3-2: File Containing Unsigned Byte Kernel *K*

byte offset ↓	contents	byte offset ↓	contents	byte offset ↓	contents
0		8	*k(0,0)*	•	•
1	*X*	9	*k(1,0)*	•	•
2		•	•	**7+*XY*-*X***	*k(X-1,Y-2)*
3		•	•	**8+*XY*-*X***	*k(0,Y-1)*
4		**7+*X***	*k(X-1,0)*	•	•
5	*Y*	**8+*X***	*k(0,1)*	•	•
6		•	•	**6+*XY***	*k(X-2,Y-1)*
7		•	•	**7+*XY***	*k(X-1,Y-1)*

A kernel-oriented procedure uses the location of the kernel's center pixel as the location in the output image for the output value. To facilitate this, kernel rows and columns always contain an odd number of pixels.

4.2.3.2 Algorithmic Specification

The *Image Understanding* benchmark consists of a series of operations to be performed, in sequence, using given image and operating parameters as initial input. Output from the benchmark consists of a table of feature values for each ROI selected. All of the operations utilize the result from the prior step as input. In addition, the ROI selection and feature extraction components utilize both the prior result and the input image, *V*, as depicted in Figure 4-15. Each component in the sequence is described individually below.

4.2.3.2.1 Morphological Filter

The morphological filter component chosen for the benchmark uses a structuring element, or kernel, *K*, that is a two-dimensional image with dimensions that are odd. Let *V* represent the input image and define the morphological operations, *erosion* ($\ominus$) and *dilation* ($\oplus$) as follows:

$$[V \ominus K] = MIN[v(x+m, y+n)] \qquad m,n \in Ros(K),\ k(m,n) \neq 0 \qquad (4.2.3.1a)$$

$$[V \oplus K] = MAX[v(x+m, y+n)] \qquad m,n \in Ros(K),\ k(m,n) \neq 0 \qquad (4.2.3.1b)$$

where each output pixel is computed at location (x, y) for a morphological kernel, *K*, which has a local region of support (Ros) that defines its geometric filtering properties with *M* columns and *N* rows. For these primitive morphological operations, MAX and MIN are computed locally for every pixel. Only nearby pixels are required to compute output pixels, specifically for the pixels in *K* that are non-zero.

For this benchmark, the morphological filter is defined as follows. As shown in Figure 4-15, *V* is the input image and *W* is the output, where

$$W = V - [(V \ominus K) \oplus K] \qquad (4.2.3.2)$$

A detailed discussion on morphology can be found in [Maragos] and [Parker 97]. The pixel values for input kernel *K* will be provided with *unsigned byte* precision and the pixels in the input image *V* will be provided with *short integer* precision as discussed in Section 4.2.3.1. The pixels in the output image *W* are required to have a minimum of *short integer* precision. Pseudo-code for the morphological filter component follows.

```
/* morphological filter component */
Get image V, kernel K
Clear images W, O1 and O2 (set to 0)
/* First calculate erosion O1 = V ⊖ K */
Loop for each pixel V(x,y) where (M-1)/2<=x<X-(M-1)/2 and
      (N-1)/2<=y<Y-(N-1)/2
      initialize minval
      Loop for each non-zero pixel k(m,n)
            minval = MIN[ minval, V(x+m,y+n)]
      End loop
      o1(x,y) = minval
End loop
/* Next calculate dilation O2 = O1 ⊕ K */
Loop for each pixel o1(x,y) where (M-1)/2<=x<X-(M-1)/2 and
      (N-1)/2<=y<Y-(N-1)/2
```

```
        initialize maxval
        Loop for each non-zero pixel k(m,n)
              maxval = MAX[ maxval, o1(x+m,y+n)]
        End loop
        o2(x,y) = maxval
End loop
/* Last output W = V -  O2 */
Loop for each pixel in w(x,y) where M-1<=x<X-M+1 and N-1<=y<Y-N+1
W(x,y) = V(x,y) - o2(x,y)
End loop
```

Note that, after processing, the valid output region will be smaller than the valid input region, since there is not enough valid input data at the outer edges to calculate valid output. In particular, for a given kernel with *M* columns and *N* rows, the outer frame of invalid data will be *a* rows at the top and bottom and *b* columns at the left and right of the image, where *a* and *b* are defined by

$$a = \frac{N-1}{2}, \qquad b = \frac{M-1}{2} \tag{4.2.3.3}$$

This effect accumulates during a process involving sequential steps, so that the final output will have an outer frame of undefined data equal in size to the sum of all the edge effects from each step within the process. This undefined outer frame should be set to the value 0 (zero).

For example, if a simple 3x3 kernel is used for the morphological filter (performing an erosion followed by a dilation), both *M* and *N* equal three. In that case, *a* and *b* are $(N-1)/2 = (M-1)/2 = 1$ pixel for each erosion or dilation. Therefore, the frame of undefined data in the output at the end of the filter is the sum $1+1=2$ pixels in width.

4.2.3.2.2 ROI Selection

The ROI selection component uses the input image *V,* and input parameters *thresholdLevel, minArea, maxRatio,* and *selectNumber,* provided in the input file as discussed in Section 4.2.3.1. The image *W* from the morphological filter component is the final input required. This component applies a threshold to image *W,* using the input *thresholdLevel,* to differentiate target pixels from background pixels. Then, target pixels are grouped together to determine how many isolated ROIs have been found. This is achieved by traversing *W* and assigning each detected target pixel to an ROI. Areas that are connected to each other are considered part of the same ROI. Two areas are declared connected if any target pixel from one area is 8-adjacent with any target pixel in the other (i.e., if they are horizontal, vertical, or diagonal neighbors). Next, an initial feature extraction is performed on these ROIs. Finally, a subset of these ROIs is selected, based on the values of the initial features. This subset of selected ROIs is used to generate the output of this component: an image *O* of detected objects or ROIs, and a list, *regions,* which includes initial features for each selected ROI.

The output *O* does not need to be an image, but does need to contain enough information so that each selected ROI is differentiated from other ROIs, and so each pixel within an ROI can be referenced. For the sake of simplicity and readability, the baseline implementation of the benchmark generates *O* in the form of an image. The depth of the values in *O* is driven by the maximum number of possible ROIs ($(2^{16}-2)$ as specified below).

Other implementations of the benchmark need not constrain O to be an image, as long as the utility of O remains. For example, instead of a complete image containing all the ROIs, each labelled with a distinct index, a subimage or *chip* could be extracted for each ROI where the chip boundaries could be defined by the smallest rectangular region containing the ROI. Then a method of obtaining the location of the ROI relative to the filtered image W must also be retained (i.e., an offset to place the chip over the proper location in W). In this manner, there would be *selectNumber* chips and offsets to specify the selected ROIs. As another example, an ROI could be specified as a list of pixel locations.

The feature extraction process is split up into two stages: 1) an initial set of features is calculated during the ROI selection component, and 2) an additional set of features is computed during the feature extraction component. This split is frequently done in deployed systems in order to minimize computation. Features in the first stage are typically not as computationally expensive as those in later stages. The features from the first stage are often used, as in this benchmark, to cull the ROI list before continuing, so that the second stage features are not computed except where necessary.

Implementations of this component are required to handle at most $(2^{16} - 2)$ number of ROIs (before going through the selection logic). Using 8-adjacent connectivity, the theoretical maximum number of distinct ROIs is approximately one quarter the number of pixels in the entire original image. Should there exist more than $(2^{16} - 2)$ ROIs in W, the ROIs beyond this number may be ignored.

Initial features for this component are extracted for each ROI that is found within W. The features are *centroid, area, perimeter, mean*, and *variance*. See [Parker 94] or [Castleman] for a detailed description of these features. The first three–*centroid, area,* and *perimeter*–measure properties of the ROI defined after the threshold is applied to W. Let T be an image, the same size as W, defined herein to be the value 1 over the target or ROI in question, and 0 elsewhere.

The *centroid* is the location that is central to the ROI, represented as T, and is computed using the following equations:

$$centroidCol = \frac{\sum t(x,y) * x}{area(T)}$$

$$x, y \in Ros(T) \qquad (4.2.3.4)$$

$$centroidRow = \frac{\sum t(x,y) * y}{area(T)}$$

where the x and y locations are relative to the pixel coordinates defined for the registered image W with column and row ranges $0 \le x < X$ and $0 \le y < Y$.

The *area* is a count of the number of target pixels contained in the ROI.

The *perimeter* is a count of the number of pixels on the target that are 8-adjacent to a background pixel. To find this value, let two pixels be defined as 8-adjacent if they are horizontal, vertical, or diagonal neighbors.

Both the thresholded version of *W* and the input image *V* are used to calculate the *mean* and *variance* features. These features are statistical measures computed for the pixel values in each ROI using the equations:

$$mean = \frac{\sum v(x,y)t(x,y)}{pixelCount} \qquad x, y \in Ros(T)$$

$$variance = \frac{\sum [v(x,y)t(x,y)]^2}{pixelCount} - mean^2 \qquad x, y \in Ros(T) \tag{4.2.3.5}$$

where the summation for the *mean* and *variance* is calculated over a single ROI, and *pixelCount* is the number of pixels in that ROI (represented above as *T*).

Once the initial features have been calculated, selection logic using these features creates a subset of selected ROIs that are retained. The three input parameters–*minArea, maxRatio,* and *selectNumber*–specify the selection criteria. The features calculated in the ROI selection component–*area, mean,* and *perimeter*–are used with these input parameters to determine whether an ROI will be selected. Any ROI with a feature that does not pass the selection criteria is removed from the ROI list. Two of the selection tests are defined below.

$$\begin{aligned} minArea &\le area \\ \frac{perimeter}{area} &\le maxRatio \end{aligned} \tag{4.2.3.6}$$

The final selection test requires ranking the list of ROIs by the value of the product (*mean* * *area*) from largest to smallest value and selecting the top *selectNumber* of ROIs that also satisfy the selection tests in Equation 4.2.3.6 above. Thus, the *selectNumber* ROIs with the largest product *mean**area*, that also satisfy the selection tests in Equation *4.2.3.6*, compose the set of selected ROIs.

Then, for the labelled ROIs, an output image, *O,* and a list, *regions,* must be constructed. Image *O* defines the shapes and locations of the selected ROIs, and *regions* contains a list of the selected ROIs including the initial features.

The output image, *O,* is required to have a minimum of *short integer* precision, to handle labels for at most $(2^{16} - 2)$ ROIs. The list shown as *regions* in Figure 4-15 must include the six features described in this section for each ROI identified in image *O.* The floating-point features–*centroidCol, centroidRow, mean,* and *variance*–are required to have a minimum of *float* precision. The other features–*area* and *perimeter*–are required to have a minimum of *integer* precision. Pseudo-code for a correct–though inefficient–implementation of the ROI selection component follows.

```
/* ROI selection component */

/* V is input image */
/* W is morphological filtered image */
/* thresholdLevel is level to use to determine target pixels */
```

```
/* minArea, maxRatio, and selectNumber are parameters
      to be used in ROI selection logic */
Get images V and W
Get parameters thresholdLevel, minArea, maxRatio, selectNumber
Clear image G (set to 0)
nt = 0
/* first threshold image W to determine target pixels
   and group pixels belonging to the same ROI, by
   marking each ROI with a unique id  */
Loop for each (x,y) in W      /* scan filtered image */
      If w(x,y) > thresholdLevel then     /* if target pixel */
            Loop for (u,v) in 8 neighbors of (x,y)
                  If g(u,v) > 0 then   /* if already tagged */
                        If g(x,y) > 0 and g(x,y) ≠ g(u,v) then
                              /* we are connecting two ROIs */
                              gt = g(u,v)
                              Loop for each (i,j) in G up to and
                                    including (x,y)
                                    If g(i,j) = gt then
                                          g(i,j) = g(x,y)
                                    Endif
                              End loop
                        Else        /* first tagged neighbor */
                              g(x,y) = g(u,v)
                        Endif
                  Endif
            End loop
            If g(x,y) = 0      /* ROI is isolated */
                  If (nt > (2^16-2)) then
                        g(x,y) = 0 /* ignore >(2^16-2) ROIs */
                  Else
                        nt = nt + 1 /* increment ROI count */
                        g(x,y) = nt /* label ROI with new id */
                  Endif
            Endif
      Endif
End loop

/* Compute initial features for each ROI */
/* F is list of initial features */
Initialize F
Loop for each ROI in G
      clear centroidCol, centroidRow, area, perimeter,
            mean, and variance
      Loop for each pixel g(x,y) in ROI
            centroidCol = centroidCol + x
            centroidRow = centroidRow + y
            area  = area  + 1
            If pixel is 8-adjacent to background pixel
                  then perimeter = perimeter + 1
            Endif
            mean = mean + v(x,y)
            variance = variance + v(x,y)*v(x,y)
      End loop
      centroidCol = centroidCol / area
      centroidRow = centroidRow / area
```

```
      mean = mean / area
      variance = (variance/area) -mean²
      add features to list F
End loop

/* use selection logic to select subset of ROIs */
order list F ranking mean*area value from largest to smallest
numROIs = 0
initialize list regions
clear image O
Loop for each ROI on list F
      If (minArea <= area) AND
         (perimeter/area <= maxRatio) AND
         (numROIs < selectNumber) Then
            numROIs = numROIs + 1
            add ROI to object image O
            add initial features to list regions
      Endif
End loop for each ROI on list F
```

4.2.3.2.3 Feature Extraction

In the final component of the sequence, additional features are calculated for the ROIs selected from the previous component. As shown in Figure 4-15, two input parameters are provided in the input file for this component. The parameters–*distanceShort* and *distanceLong*– are provided in *integer* precision as discussed in Section 4.2.3.1. The input image, *V*, the object image, *O*, and the list, *regions*, complete the inputs required for this component. The input *O* does not need to be an image, but does need to contain enough information so that each selected ROI is differentiated from each other and so each pixel within an ROI can be referenced. In this implementation, this is achieved by having *O* be an image. The depth of the values in *O* is driven by the maximum number of ROIs possible $(2^{16}-2)$.

The additional features calculated in this component give a measure of the texture of each ROI. As discussed in [Parker 97], a grey level co-occurence matrix (GLCM) contains information about the spatial relationships between pixels within an image. Statistical descriptors of the co-occurence matrix have been used as a practical method for utilizing these spatial relationships. Furthermore, [Unser] designed a method of estimating these descriptors without calculating the GLCM, instead using sum and difference histograms. The features to be calculated here are *GLCM entropy* and *GLCM energy*, and are defined in terms of a sum histogram, *sumHist*, and a difference histogram, *diffHist*. These histograms are dependent on a specific distance and direction just as the GLCM. The sum histogram, *sumHist*, is a normalized histogram of the sums of all pixels at a given distance and direction. Likewise, the difference histogram, *diffHist*, is a normalized histogram of the differences of all the pixels at a given distance and direction. The GLCM descriptors are defined as:

$$GLCM\ entropy = -\sum_i sumHist(i) * \log[sumHist(i)] - \sum_j diffHist(j) * \log[diffHist(j)] \qquad (4.2.3.7)$$

$$GLCM\ energy = \sum_i sumHist(i)^2 * \sum_j diffHist(j)^2$$

where *sumHist(i)* is the normalized sum histogram and *diffHist(j)* is the normalized difference histogram for the particular distance and direction of interest.

For this benchmark, rather than calculate these measures for all possible distances and directions, two distances are given in the input file, and four directions of interest must be used. These directions are defined as: 0°, 45°, 90°, and 135°. Therefore, the feature extraction component will compute, for each selected ROI, a total of sixteen features (two descriptors at each of two distances and four directions). Both of the descriptors – *GLCM entropy* and *GLCM energy* – are required to have a minimum of *float* precision.

The final output of the benchmark is a feature list, *features,* containing all twenty-two features (both initial and additional features), for each selected ROI. Pseudo-code for a correct–though inefficient–implementation of the feature extraction component follows.

```
/* feature extraction component */

/* V is the input image */
/* O is object image */
/* regions is list which includes initial features */
/* features is list of all features
                (initial and additional features) */
Get images V and O
Get list regions
Initialize features
numROI = 0
Loop for each ROI in O  /* scan object image */
      numROI = numROI + 1 /* keep track of number of ROIs */
      Get initial features for ROI from list regions
      Loop for distance = distanceShort and distanceLong
            /* for 0 degree direction (horizontal) */
            dx = distance
            dy = 0
            call calcDescriptors(V,O,numROI,dx,dy,energy,entropy)
            add all features(distance,0°) to features
            /* for 45 degree direction (right diagonal) */
            dx = distance
            dy = distance
            call calcDescriptors(V,O,numROI,dx,dy,energy,entropy)
            add all features(distance,45°) to features
            /* for 90 degree direction (vertical) */
            dx = 0
            dy = distance
            call calcDescriptors(V,O,numROI,dx,dy,energy,entropy)
            add all features(distance,90°) to features
```

```
            /* for 135 degree direction (left diagonal) */
            dx = - distance
            dy = distance
            call calcDescriptors(V,O,numROI,dx,dy,energy,entropy)
            add all features(distance,135°) to features
      End loop for distances
End ROI loop
End feature extraction

routine calcDescriptors( V, O, numROI, dx, dy, energy, entropy)
numlevels      = (number grey-levels in image)
numhistlevels = 2*numlevels
get array sumHist size numhistlevels, initialize to 0
get array diffHist size numhistlevels, initialize to 0

totalnumpixels = 0
Loop for each pixel in ROI numROI
      /* calculate sum and difference histograms */
      If (x,y) AND (x+dx,y+dy) legal pixel addresses in ROI Then
            Increment sumHist( [v(x,y) + v(x+dx,y+dy)] )
            Increment diffHist( numlevels+[v(x,y)-v(x+dx,y+dy)])
            Increment totalnumpixels
      Endif legal pixel address
End loop for numROI
/* normalize sumHist and diffHist */
Loop for i = 0, numhistlevels
      sumHist(i) = sumHist(i) / totalnumpixels
      diffHist(i) = diffHist(i) / totalnumpixels
End loop for i
energyS = 0
energyD = 0
entropy = 0
/* calculate descriptors from sumHist and diffHist */
Loop for i = 0, numhistlevels
      entropy = entropy - sumHist(i)*log(sumHist(i))
                        - diffHist(i)*log(diffHist(i))
      energyS = energyS + sumHist(i) * sumHist(i)
      energyD = energyD + diffHist(i) * diffHist(i)
End loop for i
energy = energyS * energyD
End routine calcDescriptors
```

4.2.3.3 Output

The output for the *Image Understanding* benchmark should be provided as a char (7-bit ASCII stored in 8-bit bytes) file where there is one line of ASCII text for each selected ROI. This entry should contain all the features calculated for that ROI: six initial features from the ROI selection component and sixteen additional features from the feature extraction component. The format for each entry is described in the following table.

Table 4.2.3-3: Output Record Specification for Each ROI

Field	Description			Type	Format
1	centroidCol			*float*	*m.dddd* E±*xx*
2	centroidRow			*float*	*m.dddd* E±*xx*
3	area			*integer*	*dddddd*
4	perimeter			*integer*	*dddddd*
5	mean			*float*	*m.dddd* E±*xx*
6	variance			*float*	*m.dddd* E±*xx*
7	DistanceShort	0°	GLCM entropy	*float*	*m.dddd* E±*xx*
8			GLCM energy	*float*	*m.dddd* E±*xx*
9		45°	GLCM entropy	*float*	*m.dddd* E±*xx*
10			GLCM energy	*float*	*m.dddd* E±*xx*
11		90°	GLCM entropy	*float*	*m.dddd* E±*xx*
12			GLCM energy	*float*	*m.dddd* E±*xx*
13		135°	GLCM entropy	*float*	*m.dddd* E±*xx*
14			GLCM energy	*float*	*m.dddd* E±*xx*
15	DistanceLong	0°	GLCM entropy	*float*	*m.dddd* E±*xx*
16			GLCM energy	*float*	*m.dddd* E±*xx*
17		45°	GLCM entropy	*float*	*m.dddd* E±*xx*
18			GLCM energy	*float*	*m.dddd* E±*xx*
19		90°	GLCM entropy	*float*	*m.dddd* E±*xx*
20			GLCM energy	*float*	*m.dddd* E±*xx*
21		135°	GLCM entropy	*float*	*m.dddd* E±*xx*
22			GLCM energy	*float*	*m.dddd* E±*xx*

One space (*char* value 32) should be used to delimit each field, and a carriage return/line feed should follow the last field for each ROI.

4.2.3.4 Acceptance Test

The software implementation will be considered successful for a given input data set when the implementation is executed for the given input data set and produces results that match with the corresponding output provided with the benchmark. Precision is discussed in Section 3.7.

4.2.3.5 Metrics

The primary metric associated with the *Image Understanding* benchmark is the total time required to run a given input data set through the Image Understanding sequence generating accurate results. A series of secondary metrics for the individual times of the processing components is also required. The time associated with a processing component is defined as the time at the beginning of one part in the flow to the beginning of the next part in the flow. For example, the time to perform the Morphological Filter component is the time it takes to apply the morphological filter to the input image V and obtain the output image W, not including time to read input image, V, and kernel, K.

4.2.3.6 Baseline Source Code

Baseline source code is available at http://www.aaec.com/projectweb/dis.

4.2.3.7 Baseline Performance Figures

Baseline performance figures are available at http://www.aaec.com/projectweb/dis.

4.2.3.8 Test Data Sets

Test data sets are available at http://www.aaec.com/projectweb/dis.

4.2.3.9 References

[Castleman] Castleman, K., *Digital Image Processing,* Prentice-Hall, 1979.

[Maragos] Maragos, P., "Tutorial on advances in morphological image processing and analysis," *Optical Engineering,* vol. 26, no. 7, pp. 623-632, July 1987.

[Parker 94] Parker, J., *Practical Computer Vision Using C,* Wiley, 1994.

[Parker 97] Parker, J., *Algorithms For Image Processing And Computer Vision,* Wiley Computer Publishing, 1997.

[Unser] Unser, M., "Sum and Difference Histograms for Texture Classification," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vol. PAMI-8, 1:118-125, 1986.

[Weeks] Weeks, A., *Fundamentals of Electronic Image Processing,* SPIE/IEEE series on imaging science & engineering, 1996.

4.2.4 Multidimensional Fourier Transform

The Fourier Transform has wide application in a diverse set of technical fields. It is utilized in image processing, convolution and deconvolution, and digital signal filtering to name a few. This benchmark attempts to measure the performance of a typical range of transforms using the candidate hardware configurations.

The DIS *Fourier Transform Benchmark* consists of applying a three-dimensional Discrete Fourier Transform (DFT) to a series of transform tests, all of which have the same number of dimensions, but may have different sizes and are repeated a specified number of times. The three-dimensional DFT is defined by

$$F(x,y,z) = \sum_{k_3=0}^{Z} \sum_{k_2=0}^{Y} \sum_{k_1=0}^{X} e^{2\pi i k_3 z/Z} e^{2\pi i k_2 y/Y} e^{2\pi i k_1 x/X} f(k_1, k_2, k_3) \qquad (4.2.4.1)$$

where f is the input complex three-dimensional array of size $X \times Y \times Z$, and F is the output forward transform of f of the same size. An individual transform test consists of a specification of the size of the input array and the number of times the input array is transformed into the output array. Note that this benchmark does not specify any recursive application of the DFT (for example, applying the transform, calculating the inverse, and applying the transform to the result). Thus, the input array, f, should not be overwritten during the calculation of the transform output, F.

The size of the three-dimensional data array to be transformed is specified by an input file (i.e., the values of X, Y, and Z), but the initial values for the array data, f, are not provided. The DFT computes the transform of a three-dimensional array of complex floats, but the computational speed or efficiency of the DFT is not data dependent. Because of this separation between data and performance, and to reduce the size of the input set, only the lengths of the three dimensions are specified. The values for the array should be randomly initialized once for each transform test. The characteristics of the random input initialization is left to the individual implementers with the conditions that the random number generator have a period larger than 2^{32} and the values generated lie within the bounds detailed for *floats* in Section 3.6 (Common Data Types). Note that almost all implementations of the standard UNIX *rand()* function satisfy these conditions.

4.2.4.1 Input

An input set for the *Fourier Transform Benchmark* is provided in a single ASCII text file as a list of transform tests. All values within the input file are integers and are white-space-delimited (here "white space" indicates carriage returns, line feeds, or spaces). The first value in the input file is an integer, which specifies the number of transform tests detailed in the file. The rest of the file consists of a series of four integers where each set of four specifies a transform test. The first three integers of a set are the lengths of the first, second, and third dimensions of the transform, respectively. The fourth integer of a set is the number of times to repeat this particular test. Table 4.2.4-1 shows a schematic of an input file where M is the number of transform tests detailed in the file and X, Y, and Z are the lengths of the first, second, and third dimensions, respectively.

Table 4.2.4-1: Fourier Transform Input Schematic

Value	Description	
integer	number of transform tests, M	
integer	length of 1st dimension, X	transform test 1
integer	length of 2nd dimension, Y	
integer	length of 3rd dimension, Z	
integer	number of iterations	
integer	length of 1st dimension, X	transform test 2
integer	length of 2nd dimension, Y	
integer	length of 3rd dimension, Z	
integer	number of iterations	
:	:	:
integer	length of 1st dimension, X	transform test M
integer	length of 2nd dimension, Y	
integer	length of 3rd dimension, Z	
integer	number of iterations	

An example of an input file is given in the table below. This example file specifies five separate transform tests: a 200x100x1 transform repeated 10,000 times, a 5000x100x1 transform repeated 5000 times, a 20,000x800x1 transform repeated 100 times, a 5000x200x2 transform repeated 250 times, and a 4000x1000x1 transform repeated 500 times.

Table 4.2.4-2: Fourier Transform Input Example

`5`				# number of transform tests, *M*
`200`	`100`	`1`	`10000`	# parameters for transform test 1
`5000`	`100`	`1`	`5000`	# parameters for transform test 2
`20000`	`800`	`1`	`100`	# parameters for transform test 3
`5000`	`200`	`2`	`250`	# parameters for transform test 4
`4000`	`1000`	`1`	`500`	# parameters for transform test 5

4.2.4.2 Algorithmic Specification

The discussion of the algorithmic specification for the DFT in this benchmark will be limited, as the amount of material freely available to the implementers is extremely large. Rather, a brief description of several FFT algorithms, with appropriate references, will be given. The focus of the discussion is on the FFT methods since almost all transform implementations are FFT rather than a direct implementation of the DFT Equation 4.2.4.1. However, any implementation which yields valid results is acceptable. The algorithm descriptions provided here are not meant as a complete listing of all available or allowable methods; implementers are encouraged to use any methods that will demonstrate the advantages of their hardware configurations. Also, a brief discussion of implementing algorithms which require dimensions of a power of two using "zero padding" is given.

The majority of FFT methods transform the original DFT into a series of subproblems which achieves a lower computational complexity[Duhamel90]. The most common subproblem decomposition is to assume that the dimensions of the input array are powers of two. This allows the summations present in Equation 4.2.4.1 to be split into two subproblems [Cooley]. The even- and odd-numbered frequencies are separated and the problem is recursively split by two until the original transform of length $N = X \times Y \times Z$ is reduced to transforms of length one, which is simply the identity operation that copies its input number to its output slot. The total process is on the order of $N \log_2 N$ and requires a bit reversal either on the input or the output depending upon the specific algorithm. Methods that do the bit-reversal then build up the transform are generally called *decimation-in-time* (DIT) or Cooley-Tukey FFT methods. Methods that manipulate the input data and then do bit-reversal on the output values are generally called *decimation-in-frequency* (DIF) or Sande-Tukey FFT methods.

The same type of reasoning can be applied, but the recursive subdivision stopped, at higher powers of two (typically four and eight with this type of algorithm called radix-4 or radix-8 methods[Ganapa]). These small transforms are done using highly optimized code, which provides a modest but appreciable performance improvement. A combination of subproblems of lengths two, four, or eight are also possible, and are generally called split-radix methods[Duhamel84].

The division of the DFT into subproblems is not limited to powers of two, but can be applied using prime numbers[Rader] and combinations of powers of two and primes with relatively sophisticated decision trees to determine the "optimal" subproblem divisions for a given problem [Frigo], [Frigo99].

The subproblem division of Equation 4.2.4.1 into powers of two requires that most of the computations, especially complex multiplications, be done in the initial stages of the algorithms for the Sande-Tukey FFT methods. However, the Cooley-Tukey methods place most of the complex computation at the final stages of the algorithms. A combination of the DIT and DIF methods with a transition stage between the domains would then lead to computation savings which is the idea behind Decimation-In-Time-Frequency methods [Saidi].

Several FFT algorithms require that the input array have dimensions that are a power of two. One method for using these algorithms when the array dimensions are not powers of two is to use a technique called "zero padding". This technique simply increases the memory size of the original array to the next power of two and initializes the extra space to zero. The numerical accuracy of the DFT algorithm is essentially unaffected by these "extra" zeros, and the result should be identical to other DFT methods. The primary trade-off is in terms of excess storage required for the technique that can become critical for large input arrays.

4.2.4.3 Output

The output of this benchmark consists of an ASCII text file indicating the "mean fractional error" of the individual transform tests. All values placed in the output file are white-space-delimited (see Section 3.6 for the definition of "white space"). The first value in the file is an integer that specifies the number of transform tests performed and should match the corresponding value from the input file. The float values for the "mean fractional error", ε_{mfe}, for each test are then listed in the order they were performed. Table 4.2.4-3 shows a schematic of an output file.

Table 4.2.4-3: Fourier Transform Output Schematic

Value	Description
integer	number of transform tests, M
float	mean fractional error of test 1
float	mean fractional error of test 2
:	:
float	mean fractional error of test M

The "mean fractional error", ε_{mfe}, is defined to be

$$\varepsilon_{mfe} = \frac{1}{XYZ} \sum_{z} \sum_{y} \sum_{x} \frac{2\left|f_{xyz} - \hat{f}_{xyz}\right|}{\left|f_{xyz}\right| + \left|\hat{f}_{xyz}\right| + \varepsilon} \tag{4.2.4.2}$$

where f is the original input to the transform, $\hat{f}$ is the inverse of the transform of f, i.e.,

$$\hat{f} = F^{-1}(f) = \frac{1}{XYZ} \sum_{k_3=0}^{Z} \sum_{k_2=0}^{Y} \sum_{k_1=0}^{X} e^{-2\pi i k_3 z / Z} e^{-2\pi i k_2 y / Y} e^{-2\pi i k_1 x / X} F(k_1, k_2, k_3) \tag{4.2.4.3}$$

which differs from Equation 4.2.4.1 by simply changing the sign within the exponents and dividing the result with the total size of the transform. The variable ε within Equation 4.2.4.2 is a small value used to prevent division by zero. The value ε_{mfe} is then a measure of the perturbation of the transformed data from the original. Note that it is not necessary to implement an inverse DFT to calculate $\hat{f}$. The array $\hat{f}$ can be calculated from the identity,

$$\hat{f} = F^{-1}(f) = F^{*}(f^{*}) \qquad (4.2.4.4)$$

where * indicates complex conjugation.

4.2.4.4 Acceptance Test

A given input set will be considered successfully executed when each transform test successfully passes the output provided with the benchmark. An individual transform test is considered successfully executed when the value for ε_{mfe} is less than or equal to the value. An input set is considered successfully executed when all of the individual transform tests pass this same level of required accuracy.

4.2.4.5 Metrics

There are three metrics for this benchmark. The first, and primary, is the total time required to complete the input set. This should include the time for each transform test as well as the I/O time required to load the randomly generated input and output the result. The total time should not include the time necessary for the generation of the random data. The second metric is the time required to complete the individual transform tests. Again, this time should include any I/O time for loading of data and output of results. The third metric measures the "mflops" [Johnson] of the individual transform tests. The "mflops" for a given transform is defined to be

$$"mflops" = \frac{5(X \times Y \times Z)\log_2(X \times Y \times Z)}{(\text{time for one DFT in } \mu\text{s})} \qquad (4.2.4.5)$$

where X, Y, and Z are the lengths of the first, second, and third dimensions, respectively. The rational behind using this metric is to provide a reasonable comparison between different architectures, implementations, and transform sizes. Note that the "mflops" is not the MFLOPS (millions of floating-point operations per second), but an estimate of that value which assumes a common baseline number of operations for any implementation as

$$5(X \times Y \times Z)\log_2(X \times Y \times Z) + \vartheta(N) \qquad (4.2.4.6)$$

which is the radix-2 Cooley-Tukey FFT[Cooley]. This third metric is common in the FFT literature and for more discussion of the reasoning behind the metric, the reader is referred to [Johnson].

4.2.4.6 Baseline Source Code

Baseline source code is available at http://www.aaec.com/projectweb/dis.

4.2.4.7 Baseline Performance Figures

Baseline performance figures are available at http://www.aaec.com/projectweb/dis.

4.2.4.8 Test Data Sets

Test data sets are available at http://www.aaec.com/projectweb/dis.

4.2.4.9 References

[Duhamel90] Duhamel and Vetterli, "Fast Fourier transforms: a Tutorial Review and State of the Art," *Signal Processing*, vol. 19, pp.259-299, April 1990.

[Cooley] Cooley and Tukey, "An Algorithm for Machine Computation of Complex Fourier Series,," *Math. Comp.*, vol. 19, pp.297-301, April 1965.

[Ganapa] Ganapathiraju, Hamaker, Picone and Skjellum, "Analysis and Characterization of Fast Fourier Transform Algorithms," MS State High Performance Computing Laboratory, Oct. 1997.

[Duhamel84] Duhamel and Hoolomann, "Split Radix FFT Algorithm," *Electronic Letters*, vol. 20, pp.14-16, Jan 1984.

[Rader] Rader, "Discrete Fourier Transforms when the Number of Data Samples is Prime," *Proc. of the IEEE*, vol. 56, pp.1107-1108, June 1968.

[Frigo] Frigo and Johnson, The FFTW web page, http://theory.lcs.mit.edu/~fftw

[Frigo99] Frigo, "A Fast Fourier Transform Compiler," MIT Laboratory for Computer Science, Feb. 16, 1999.

[Saidi] Saidi, "Decimation-In-Time-Frequency FFT Algorithm," *Proc. of International Conference on Acoustics, Speech, and Signal Processing*, vol. III, pp.453-456, Adelaide, Australia, April 1994.

[Johnson] Frigo and Johnson, The BenchFFT web page, http://theory.lcs.mit.edu/~benchfft

4.2.5 Data Management

The Data Management Benchmark measures application-level timing performance of typical DBMS. This benchmark focuses on index management and ad hoc or content-based queries since these two areas are the primary weaknesses of traditional DBMS.

The benchmark is implemented as a simplified object-oriented database with an R-Tree indexing scheme. The R-Tree index is a height-balanced containment structure that uses multidimensional hyper-cubes as keys. The intermediate nodes are built up by grouping all of the hyper-cubes at the lower level. The grouping hyper-cube of the intermediate node completely encloses all of the lower hyper-cubes, which may be points. The system must respond to a set of command operations: *Insert*, *Delete*, and *Query*, queries being either key-based or content-based. The commands are to be issued to the system in a batch form as a data set.

The *Insert* command operation places a new data object into the database with the specified attribute values. Each *Insert* command contains all the information contained by the data object, including the hyper-cube key and list of non-key attribute values.

The *Query* command operation searches the database and returns all data objects that are consistent with a list of input data attribute values. The input attribute values can specify attribute values which are *key*, *non-key*, or both. A data object is consistent with the *Query* when the input values intersect the stored values of the data object.

The *Delete* command operation removes all objects from the database that are consistent with a list of input data attribute values. The types and conditions of the input attribute list, as well as the description for consistency, are the same as for the *Query* operation.

4.2.5.1 Input

The input for each test of this benchmark consists of one data set. All of the data sets share a common format. Each set is a 8-bit ASCII character file and consists of a series of sequentially issued commands delimited by a carriage return, i.e., each line of the file represents a separate command. Table 4.2.5-1 gives the command operations, the character code used to designate the command, the data placed after the command code on the rest of the line, the return expected from the application, and a brief description of the operation.

Table 4.2.5-1: Command Operations

Command	Code	Line Elements	Return	Description
Initialization	0	Fan Size	NULL	Initializes the index by specifying the fan of the tree.
Insert	1	Object Type Key Attribute Key Attribute : Key Attribute Non-Key Attribute Non-Key Attribute : Non-Key Attribute	NULL	Insert new entry into database. See below for discussion of the Object Type and key and non-key attributes.
Query	2	Attribute Code Attribute Value Attribute Code Attribute Value : Attribute Code	Data Object List	Return all data objects that are consistent with the input attributes specified. Note that attribute codes and values always appear as pairs.

		Attribute Value		
Delete	3	Attribute Code Attribute Value Attribute Code Attribute Value : Attribute Code Attribute Value	NULL	Delete all data objects that are consistent with the input attributes specified. Note that attribute codes and values always appear as pairs.

Each data object has a set of attributes, where the first eight attributes are used by the R-Tree index as the key and represent two points that specify a hyper-cube. Each point consists of four 32-bit IEEE-formatted floating-point numbers denoting a four-dimensional point in Euclidean space as the T-position, X-position, Y-position, and Z-position. Thus, the index key, which consists of a "lower" and "upper" point, is eight 32-bit floating-point numbers. Note that a point in hyper-space can be thought of as time (T) and a three-dimensional point (X,Y,Z), but this is immaterial to this benchmark.

The total number of attributes assigned to a data object is the sum of the key and non-key attributes. The number of non-key attributes for a given data object is determined by the Object Type, and is given in the table below. The Object Type used by the *Insert* command specifies which of the three types of objects (Small, Medium, and Large) is being inserted by the operation. Table 4.2.5-2 gives the character/byte code and the number of non-key attributes for each data object type

Table 4.2.5-2: Data Object Types

Object Type	Code	No. of Non-Key Attributes
Small	1	10
Medium	2	17
Large	3	43

Data objects differ by the number of non-key attributes assigned to each. Each non-key data attribute has an identical format that primarily consists of an 8-bit NULL-terminated ASCII character sequence of maximum length 1024. Table 4.2.5-2 gives the number of non-key attributes assigned to each object type. The maximum sizes for the Small (*overhead* + 10 * 1024), Medium (*overhead* + 17 * 1024), and Large (*overhead* + 43 * 1024) data objects are known beforehand, but the total size of the database is not determined until the input is set. The database should be able to handle all three types of data object, in any permutation. Note that the object type specification is placed at the beginning of the *Insert* command as a convenience, since the number of attributes can be determined by reading until the next carriage return, i.e., the end of the command.

The *Delete* and *Query* commands each reference a specified attribute by means of an Attribute Code. The following table gives the attribute code sequence for both the key and non-key attributes. Also, each attribute is assigned a type and, if applicable, a name and units.

Table 4.2.5-3: Attribute Codes and Descriptions

		Attribute Code	**Name**	**Type**
Key Attributes	Lower Point	0	T	float
		1	X	float
		2	Y	float
		3	Z	float
	Upper Point	4	T	float
		5	X	float
		6	Y	float
		7	Z	float
Non-Key Attributes	Small	8	property	char string
		9	property	char string
		:	:	:
		16	property	char string
		17	property	char string
	Medium	18	property	char string
		19	property	char string
		:	:	:
		23	property	char string
		24	property	char string

	Large	25	property	char string
		26	property	char string
		:	:	:
		49	property	char string
		50	property	char string

Only the *Query* commands result in a response from the database. The response is the set of data objects that are appropriate for the corresponding *Query*. A description of the response is given in the Output section of this benchmark specification.

The formal definition of each command input line is described in the following sections. Each command line represents separate pieces of data, which are ASCII space-delimited unless otherwise stated. References to integers and floats indicate 32-bit IEEE standards. The first piece of data for every command line is the command code, which indicates the specific operation. The rest of the line is relative to the command type and is described in detail below.

4.2.5.1.1 Initialization

The *Initialization* command appears only once per data set and is always the first command. Only two pieces of information is provided for the command where the first is the command type, which in this case is '0'. The second piece of data is the fan size, which is an integer. A diagram of the *Initialization* command line is given below:

int	*int*
code	fan size

4.2.5.1.2 Insert

The *Insert* command is different from the *Delete* and *Query* commands, in that the *Insert* command does not reference data attributes by the appropriate attribute code given in Table 4.2.5-3. The *Insert* command does not need to specify the attribute code, since all attributes are provided in the command line and are in the proper order. Thus, the *Insert* command input line does not use the attribute codes in order to reduce the size of the input data sets and to simplify the command line input.

The first piece of data in the command line is the command type, which is '1' for the *Insert* operation. The next piece of data represents the object type, and can be the character 1, 2, or 3, for Small, Medium, or Large, respectively. The next eight pieces of data make up the index key for the object as the floating-point values for T, X, Y, and Z, for the "lower" and "upper" hyper-points, respectively. The remaining data are the individual non-key attribute character sequences of the new object. Each attribute is space-delimited, and is of variable length. The number of attributes on the line is dependent upon the object type being read, and is given in Table 4.2.5-2. The maximum size for any attribute on the command line is 1024, although in practice the attributes will be smaller. A diagram of the *Insert* command line is given below:

int	*int*	*float float float float*	*float float float float*	*char char* ⋯ *char*
code	type	key attributes		non- key attributes

4.2.5.1.3 Query

The *Query* command returns all data objects within the current database that are consistent with the provided key and non-key attribute values. A data object is consistent when the object's attribute values "intersect" with the *Query* attribute values. The definition of intersection is different for the key and non-key attributes. The attributes that make up the index key are hyper-cubes and the definition for an intersection is an intersection of the respective hyper-cubes, i.e., an intersection occurs whenever the input hyper-cube and the stored hyper-cube share any of the hyper-space. The non-key attributes consist of character sequences and the definition of an intersection is when any part of the stored character sequence matches the entire input sequence.

The first piece of data is the command code, which is '2' for *Query*. The rest of the command line is a list of attribute code and value pairs. A diagram of the *Query* command line is given below:

int	*int*	*float / char*	⋯	*int*	*float / char*
code	attribute code	attribute value		attribute code	attribute value

The number of attribute code-value pairs ranges from 1 to 50, where an attribute code is never repeated in a single command line. The attribute value type depends upon the attribute code where an attribute code between zero and seven indicates a float and an attribute code between eight and 50 indicates a character sequence. It is possible that a *Query* will specify an attribute code that is not applicable to a specific data object. For example, any attribute code greater than 17 for a Small object, or any attribute code greater than 24 for a Medium object. The query search values for these cases should be ignored, and should not prevent the candidate data object from inclusion in the *Query* solution.

A key query, a search which uses the R-Tree index to search the database, requires a full index key, i.e., all eight floating point values specifying the search hyper-cube. The *Query* command input line need not contain all eight values for the search. If so, the search hyper-cube uses "wild-card" values for the rest of the search hyper-cube that match all possible stored values. An example of an incomplete key *Query* is

2 3 0.0 7 10.0

which searches the current database for all data objects which were between the Z-positions of 0.0 and 10.0 for any values for the T-, X-, and Y-positions. Similarly, ad hoc queries also use wild-card values for missing values of the search hyper-cube.

4.2.5.1.4 Delete

The *Delete* command removes all data objects in the database that are consistent with the provided attributes. A data object is consistent when the object's attribute values "intersect" with the *Query* attribute values. The definition of intersection is different for the key and non-key attributes. The attributes that make up the index key are hyper-cubes and the definition for an intersection is an intersection of the respective hyper-cubes, i.e., an intersection occurs whenever the input hyper-cube and the stored hyper-cube share any of the hyper-space. The non-key attributes consist of character sequences and the definition of an intersection is when any part of the stored character sequence matches with the input sequence. This description of consistency is identical to the one given for the *Query* command in the previous section.

The first piece of data is the command code, which is '3' for the *Delete* operation. The rest of the command line is a list of attribute code and value pairs and is identical to the *Query* command given in the previous section. A diagram of the *Delete* command line is given below.

int	*int*	*float / char*	...	*int*	*float / char*
code	attribute code	attribute value		attribute code	attribute value

The description of the attribute code-value pairs and the use of wild-card values is identical to the *Query* command given in the previous section.

4.2.5.2 Algorithmic Specification

The database consists of various combinations of the three types of data objects. The algorithm requires the maintenance of an R-Tree index structure. The program shall respond to the command operations: *Insert*, *Query*, and *Delete*. This section has three parts: the data object description, R-Tree index structure and description, and R-Tree variant discussion.

4.2.5.2.1 Data Object Description

Each entry in the database will be one of three types as discussed in section 4.2.5.1. A data object has a set list of attributes, which are the sum of the key and non-key attributes. The first eight attributes represent the index key and is specified as eight 32-bit IEEE floating-point numbers representing the T, X, Y, and Z-positions of both the "lower" and "upper" points of a hyper-cube, respectively. Finally, each object has a constant number of attributes or parts. A non-key data object attribute is an 8-bit NULL-terminated ASCII character sequence of maximum length 1024. The number of attributes assigned to each data object type is given in Table 4.2.5-2. The attributes for a given data object reference each other as a single-linked list and the data object holds a reference to the first attribute in the list which is defined as the head. Separate indices that would use the non-key attributes are not permitted for this benchmark. Thus, a *Query* operation which contains no key search information will search the entire database for consistent entries.

4.2.5.2.2 R-Tree

This benchmark requires the implementation of a simplified object-oriented database and an attendant R-Tree indexing structure. A general R-tree has the following properties:

1. All leaves are at the same level (height-balanced).

2. Every node contains between kM and M index entries unless it is the root. (M is the order of the tree).

3. For each entry in an intermediate node, the sub-tree rooted at the node contains a hyper-cube if and only if the hyper-cube is "covered" by the node, i.e., containment.

4. The root has at least two children, unless it is a leaf.

This benchmark requires the R-Tree structure be maintained during execution of the database implementation. However, the particular method used to maintain the R-Tree (search, tree compacting, etc.) is left to the user.

This section gives a brief description of the R-Tree algorithm and two variants. One variant allows for concurrency assurance without a full index list update (R-Link tree); the other seeks to minimize the overlap of the R-Tree with the offset of increasing the tree's height (R+-Tree). The user is encouraged to implement the standard R-Tree, variant described here, or other variant, as is most suitable for the hardware being tested. Descriptions given here are for illustrative purposes; they are not intended to dictate implementation strategy.

The R-Tree index provides a multi-dimensional data indexing scheme. It is a direct extension of the B-Tree in k dimensions (where k = 4 for this benchmark). The structure is a height-balanced containment tree, which consists of intermediate and leaf nodes. The data objects are stored as leaf-nodes (requiring four-dimensional position information). The intermediate nodes are built up by grouping all of the hyper-cubes at the lower level. The grouping hyper-cube of the intermediate node completely encloses all of the lower hyper-cubes and/or points. An example is the placement of rectangles in a Cartesian plane given in Figure 4-17.

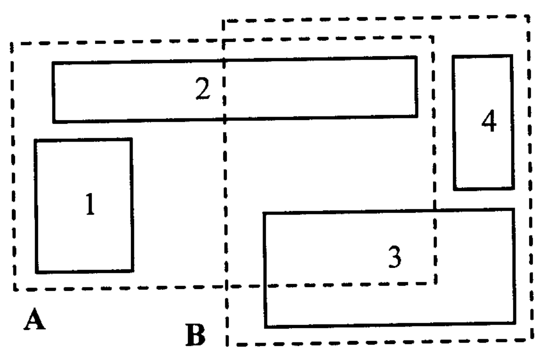


Figure 4-17: R-Tree 2D Example

This layout of rectangles would produce an index given in Figure 4-18.

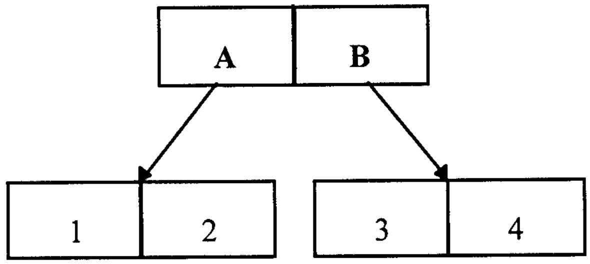


Figure 4-18: R-Tree Structure Example

The first command operation to be detailed is *Insert* . The *Insert* is the method used to place new data objects into the index and is the primary index management method, and thus the most complex. The command operation *Insert* for a generic R-Tree is given in Figure 4-19.

Method Insert(R, E)

Input: An R-Tree rooted at R and new data object with input hyper-cube E

Output: The new R-Tree after insertion of data object.

Method: Find where object should go and add to leaf nodes, splitting if necessary.

1. **Find leaf for insertion**. If R is not a leaf, recursively descend R and find L which is defined as the leaf node of R which gives the minimum penalty. The penalty of "change in area" proposed by Guttman is defined as the difference between the union of the hyper-cubes L and E and the area of L. The union of two hyper-cubes is itself a hyper-cube which minimally spans its components.
2. **Insert**. If L is not full, install E on L. Otherwise split L. Splitting L consists of separating L into two groups according to a similar "change in area" penalty. One group is placed on the new node, and the other is Inserted into the parent, splitting again if necessary.
3. **Adjust Keys**. Check the immediate parent of new node. If the key is already accurate or if there is no parent, stop. Otherwise, modify the parent to be the union of its children. Recursively ascend tree until root, R.

Figure 4-19: Insert

The "change in area" penalty is only one of several methods in choosing the leaf for insertion and for splitting. The user is referred to [Guttman], [Kornacker], and [Sellis], for a sampling of the different methods.

The second command operation detailed is the *Query* command, which is described in Figure 4-20. The *Query* command recursively descends all paths of the R-Tree which are consistent with the input search key returning all data objects which are consistent with the same search key.

Method Query (R, K, A)

Input: An R-Tree rooted at R, search key K, and non-key search values A

Output: The set of objects which are consistent with search key K.

Method: Recursively descend all paths of R which are consistent with K.

1. **Search**. Check each subtree of R to see if K is consistent. If so, search on subtree until leaf
2. **Check Key Attributes.** If current node is a leaf, check if K is consistent with data object. If so, add data object to solution set
3. **Check Non-Key Attributes.** Remove all entries in current solution set which are not consistent with non-key attributes of input values A.
4. **Return**. Return complete solution set

Figure 4-20: Query

The *Query* command detailed in Figure 4-20 is for key, non-key, and ad hoc queries. Note that a non-key query will check the entire database for all consistent objects. The method given in Figure 4.6 will work for a non-key query but will yield poor performance. Because of the default "wild-cards" for the non-specified search hyper-cube, the second step of the method will return the entire database as a list, which will then be searched by the third step. The creation of that list using the R-Tree index is not efficient and the benchmark implementors are encouraged to have auxiliary lists or parallel searches to improve the performance for non-key queries.

The final command operation detailed is the *Delete* operation, described in Figure 4-21. The purpose of the *Delete* command is to measure the performance of index management when entries are removed. The *Delete* command presented here uses the *Query* operation to determine the data objects that need to be removed, and so, the *Delete* performance is also a measure of the *Query* performance.

Method Delete (R, K, A)	
Input:	An R-Tree rooted at R, search key K, and non-key search values A
Output:	The new R-Tree after deletion of all consistent data objects.
Method:	Remove all data objects consistent with both K and A.

1. **Search** . Query index for all entries, L, consistent with K and A.
2. **Delete** . Remove all entries in L from R
   2.1. **Remove:** Remove entry in L from parent leaf, P.
   2.2. **Condense** . Recursively ascend tree, from P, adjusting the keys to minimize the penalty until the root, R.
3. **Clean-up:** If the root node has only one child, make child the new root.

Figure 4-21: Delete

4.2.5.2.3 R-Tree Variants

The user is constrained to implement the R-Tree structure as the indexing scheme for this benchmark application. However, the user is encouraged to select any implementation or variant of the R-Tree algorithm. Two variants, which may improve performance for a specific hardware architecture, are the R-link tree and the R+-Tree; these are discussed here.

The R-link tree variant uses a technique corresponding to the B-link tree to develop a scheme that does not require parent node locking for concurrent operations on the tree. Two differences from the base R-Tree are introduced for R-link trees. The first requires that all nodes within a level are right-linked together for a singly linked list. The second difference adds a logical sequence number (LSN) to each node which is unique within the tree/partition. The R-link algorithm uses the LSN to provide a mechanism for determining when an operation's understanding of a given node is obsolete, i.e., a node split has occurred. If a split has occurred, the right-link is used to traverse the tree until a correct or expected LSN is found.

The R+-Tree eliminates overlap by reducing any overlapping hyper-cubes into sub-cubes and redistributing the tree. This provides a marked increase in search performance. The increase is offset by a more complicated index maintenance and by an increase of approximately 10% for the space required for the index.

4.2.5.3 Output

The output of the database will be the responses to each *Query* operation.

The response to a *Query* operation consists of a set of data objects that are consistent with the *Query*. Each data object in a response is represented as the list of its attributes in order defined by Table 4.2.5-3. The list of attributes shall be written to an 8-bit ASCII character file where each attribute is space delimited and with each list carriage return delimited. The format is very similar to the *Insert* operation format for the input data sets with the only difference being the omission of the command code. The set of data objects returned by a *Query* must be placed in the output file continuously, i.e., in adjacent lines, although the order of the set is not constrained.

4.2.5.4 Acceptance Test

A given data set will be considered successfully executed when the command operation query responses match with the corresponding data set query responses provided by the baseline results.

4.2.5.5 Metrics

The primary metric associated with the Data Management benchmark is total time for accurate completion of a given input data set. A series of secondary metrics are the individual times of the command operations: *Insert*, *Delete*, and *Query*. Best, worst, and average times should be reported for all operations for each data set.

The time for a non-response command operation to complete is defined as the difference between the time immediately before the command is placed in the database input queue and the time immediately before the next command is placed in the same input queue. This time difference is essentially the rate at which each line of the input data set is read. This definition is applied to the *Insert* and *Delete* command operations. The time for a *Query* command operation to complete is defined as the difference between the time immediately before the command is placed in the input queue to the time immediately after the response is placed in the output queue.

4.2.5.6 Baseline Source Code

Baseline source code is available at http://www.aaec.com/projectweb/dis.

4.2.5.7 Baseline Performance Figures

Baseline performance figures are available at http://www.aaec.com/projectweb/dis.

4.2.5.8 Test Data Sets

Test data sets are available at http://www.aaec.com/projectweb/dis.

4.2.5.9 References

[Guttman] Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOID*, pp. 47-57, June 1984.

[Kornacker] Kornacker, Banks, "High-Concurrency Locking in R-Trees," *Proceedings of 21st International Conference on Very Large Data Bases,* pp. 134-145, September 1995.

[Sellis] Sellis , Roussopoulos, and Faloutsos, "The R+-Tree: A Dynamic Index for Multi-Dimensional Objects," *Proc. 13th International Conference on Very Large Data Bases*, pp. 507-518, Brighton, September 1987.

5. CONTACT INFORMATION

For questions about...	Contact...
Data-Intensive Systems program	Dr. José Muñoz DARPA / ITO 3701 North Fairfax Drive Arlington, VA 22203
This Document Benchmarks Procedures Input/Output Data Baseline Performance Reporting of Results Multidimensional Fourier Transform Benchmark Image Understanding Benchmark Data Management Benchmark	Joseph F. Musmanno Atlantic Aerospace Electronics Corporation 470 Totten Pond Road Waltham, MA 02451 Telephone: 781-890-4200x3218 Fax: 781-890-0224 Email: joe@aaec.com
Method of Moments Benchmark	Joseph W. Manke, Ph.D. The Boeing Company PO Box 3707 MC 7L-21 Seattle, WA 98124-2207 Telephone: 425-865-3163 Fax: 425-865-2966 Email: joseph.w.manke@boeing.com
Ray-Tracing Benchmark	Jon W. Harris ERIM International, Inc. PO Box 134008 Ann Arbor, MI 48113-4008 Telephone: 734-994-1200x3313 Fax: 313-994-5124 Email: jharris@erim-int.com

6. REFERENCES

DIS Program

[DARPA] DARPA/ITO website http://www.darpa.mil/ito.

[Muñoz] Dr. José Muñoz, presentation at Data-Intensive Systems Principal Investigators' meeting, 1 October, 1998, http://www.darpa.mil/ito/research/pdf_files/dis_approved.pdf.

Benchmarking

[Honeywell] Honeywell, Inc., *Benchmarking Tools and Assessment Environment for Configurable Computing: Benchmark Definition Methodology Document*, submitted to USA Intelligence Center and Fort Huachuca under contract number DABT63-96-C-0085, 19 February 1998.

[Weems] Weems, Riseman, and Hanson, The DARPA Image Understanding Benchmark for Parallel Computers, *Journal of Parallel and Distributed Computing*, 11, 24 January 1991.

[ASCII] *Information Systems - Coded Character Sets - 7-bit American National Standard Code for Information Interchange*, ITI (NCITS), ANSI x3.4-1986 (R1997).

[Float] *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std* 754-1985 (IEEE 754), published by the Institute of Electrical and Electronics Engineers, Inc, 345 East 47th Street, New York, NY 10017, 1986.

Method of Moments

[Rokhlin-1] V. Rokhlin, "Diagonal Forms of Translation Operators for the Helmholtz Equation in Three Dimensions", Research Report YALEU/DCS/RR-894, Dept. of Comp. Sci., Yale Univ., March, 1992.

[Rokhlin-2] R. Coifman, V. Rokhlin and S. Wandzura, "The Fast Multipole Method for the Wave Equation: A Pedestrian Prescription", IEEE Antennas and Propagation Magazine, 35, No. 3, June 1993, pp. 7-12.

[Dembart-1] B. Dembart and E. L. Yip, "A 3-d Fast Multipole Method for Electromagnetics with Multiple Levels", ISSTECH-97-004, The Boeing Company, December, 1994.

[Dembart-2] M. A. Epton. and B. Dembart, "Multipole Translation Theory for the 3-D Laplace and Helmholtz Equations", SIAM J. Sci. Comput. 16, No. 4, pp. 865-897, July, 1995.

[Dembart-3] M. A. Epton and B. Dembart, "Low Frequency Multipole Translation Theory for the Helmholtz Equation", SSGTECH-98-013, The Boeing Company, August, 1998.

[Dembart-4] M. A. Epton and B. Dembart, "Spherical Harmonic Analysis and Syntheses for the Fast Multipole Method", SSGTECH-98-014, The Boeing Company, August, 1998.

[Saad] Yousef Saad, "Iterative Methods for Sparse Linear Systems", PWS Publishing Company, Boston, MA, 1996.

Ray Tracing References

[1] K. Bouatouch and T. Priol. Parallel Space Tracing: An Experience on an IPSC Hypercube. In N. Magnenat-Thalmann and D. Thalmann, editors, *New Trends in Computer Graphics (Proceedings of CG International '88)*, pages 170–187, New York, 1988. Springer-Verlag.

[2] J. G. Cleary, B. M. Wyvill, G. M. Birtwistle, and R. Vatti. Multiprocessor Ray Tracing. *Computer Graphics Forum*, pages 3–12, 1986.

[3] F. C. Crow, G. Demos, J. Hardy, J. McLaugglin, and K. Sims. 3d image synthesis on the connection machine. In *Proceedings Parallel Processing for Computer Vision and Display*, Leeds, 1988.

[4] M. A. Z. Dipp′ e and J. Swensen. An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis. *ACM Computer Graphics*, 18(3): 149–158, Jul 1984.

[5] S. A. Green and D. J. Paddon. Exploiting Coherence for Multiprocessor Ray Tacing. *IEEE Computer Graphics and Applications*, pages 12–27, Nov 1989.

[6] H. Kobayashi, S. Nishimura, H. Kubota, T. Nakamura, and Y. Shigei. Load Balancing Strategies for a Parallel Ray-Tracing System Based on Constant Subdivision. *The Visual Computer*, 4(4): 197–209, 1988.

[7] A. J. F. Kok. *Ray Tracing and Radiosity Methods for Photorealistic Image Synthesis*. PhD thesis, Delft University of Technology, Jan 1994.

[8] T. T. Y. Lin and M. Slater. Stochastic Ray Tracing Using SIMD Processor Arrays. *The Visual Computer*, 7:187–199, 1991.

[9] D. J. Plunkett and M. J. Bailey. The vectorization of a ray-tracing algorithm for Improved Execution Apeed. *IEEE Computer Graphics and Applications*, 5(8):52–60, aug 1985.

[10] T. Priol and K. Bouatouch. Static Load Balancing for a Parallel Ray Tracing on a MIMD hypercube. *The Visual Computer*, 5:109–119, 1989.

[11] E. Reinhard. Hybrid Scheduling for Parallel Ray Tracing. TWAIO final report, Delft University of Technology, Jan 1996.

[12] I. D. Scherson and C. Caspary. A Self-Balanced Parallel Ray-Ttracing Agorithm. In P. M. Dew, R. A. Earnshaw, and T. R. Heywood, editors, *Parallel Processing for Computer Vision and Display*, Volume 4, pages 188–196, Wokingham, 1988. Addison-Wesley Publishing Company.

[13] L. S. Shen, E. Deprettere, and P. Dewilde. A New Space Partition Technique to Support a Highly Pipelined Parallel Architecture for the Radiosity Method. In

Advances in Graphics Hardware V, proceedings Fifth Eurographics Workshop on Hardware. Springer-Verlag, 1990.

[14] E. R. Frederik, W. Jansen . Rendering Large Scenes Using Parallel Ray Tracing. *Parallel Computing*, pages 873-885, 1997

[15] T. Wilson, N. Doe. Acceleration Schemes for Ray Tracing. Report Number: CS-TR-92-22, Department of Computer Science, University of Central Florida, September 1992.

[16] R.L. Cook, T. Porter, L. Carpenter. Distributed Ray Tracing. *Computer Graphics* (Proceedings of SIGGRAPH 1984), 18(3), 137-145, July 1984.

[17] R.L. Cook. Stochastic sampling in computer graphics, ACM Transaction in Graphics 5(1), 51-72, January 1986.

[18] A. S. Glassner (Editor), *An Introduction to Ray Tracing*, Academic Press 1989.

[19] Ray Tracing Bibliography, http://www.cm.cf.ac.uk/Ray.Tracing/RT.Bibliography.html

Simulated SAR References

[1] D.J. Andersh, M. Hazlett, S.W. Lee, D.D. Reeves, D.P. Sullivan and Y. Chu, "Xpatch: A high fre-quency electromagnetic-scattering prediction code and environment for complex three-dimensional objects," *IEEE Antennas & Propagation. Magazine,* vol. 36, pp.65-69, 1994.

[2] J. Baldauf, S.W. Lee, L. Lin, S.K. Jeng, S.M. Scarborough, and C.L. Yu, "High frequency scattering from trihedral corner reflectors and other benchmark targets: SBR vs. experiment," *IEEE Transacrions on Antennas and Propagation,* vol. 39, pp. 1345-1351, 1991.

[3] R. Bhalla and H. Ling, *Image-domain ray tube integration formula for the shooting and bouncing ray technique,* University of Texas Report, NASA Grant NCC 3-273, July 1993.

[4] R. Bhalla and H. Ling, "A fast algorithm for signature prediction and image formation using the shooting and bouncing ray technique," to appear in *IEEE Transactions on Antennas and Propagation,* 1995.

[5] G. Franceschetti, M. Migliaccio, D. Riccio, and G. Schirinzi, "SARAS: A Synthetic Aperture Radar (SAR) Raw Signal Simulator," *IEEE Transactions on Geoscience and Remote Sensing,* Vol. 30, No. 1, January 1992.

[6] G. Franceschetti, M. Migliaccio, and D. Riccio, "SAR Raw Signal Simulation of Actual Ground Sites in Terms of Sparse Input Data," *IEEE Transactions on Geoscience and Remote Sensing,* Vol. 32, No. 6, November 1994.

[7] D.E Herrick and I.J. LaHaie, *SRIM Polarimetric Signature Modeling,* ERIM IR&D Final Report 675805-1-F, December 1988.

[8] D.E Herrick and B.J. Thelen, "Computer Simulation of Clutter in SAR Imagery," *Proceedings of the Progress in Electromagnetics Research Symposium,* Cambridge, MA, July 1991

[9] D.E Herrick, "Computer Simulation of Polarimetric Radar and Laser Imagery," in *Direct and Inverse Methods in Radar Polarimetry,* W.-M. Boerner *et al.* (eds), Klumer Academic Publishers, The Netherlands 1992.

[10] D.E Herrick, M.A. Ricoy, and W.D. Williams, "Modeling of Foliage Effects in UHF SAR", *Proceedings qfthe Ground Target Modeling and Validation Conference,* Houghton, MI, August 1994.

[11] D.E Herrick, M.A. Ricoy, and W.D. Williams, "Synthesizing SAR Signatures of Ground Vehicles with Complex Scattering Mechanisms", *Proceedings of the Ground Target Modeling and Validation Conference,* Houghton, MI, August 1994.

[12] E.R. Keydel, D.E *Henick,* and W.D. Williams, "Interactive Countermeasures Design and Analysis Tool," *Proceedings of the Ground Target Modeling and Validation Conference,* Houghton, MI, August 1994.

[13] S.W. Lee and D.J. Andersh, *On Nussbaum Method for Exponential Series,* Electromagnetic Laboratory Technical Report ARTI-92-11, University of Illinois, Urbana, November, 1992.

[14] H. Ling, R.C. Chou, and S.W. Lee, "Shooting and Bouncing Rays: Calculating the RCS of an arbitrarily shaped cavity," *IEEE Transactions on Antennas and Propagation,* vol. 37, pp. 194-05, 1989.

[15] J.M. Nasr and D. Vidal-Madjar, "Image Simulation of Geometric Targets for Spaceborne Synthetic Aperture Radar", *IEEE Transactions on Geoscience and Remote Sensing,* Vol. 29, No. 6, November 1991.

[16] N.D. Taket, S.M. Howarth, and R.E. Burge, "A Model For the Imaging of Urban Areas by Synthetis Aperture Radar," *IEEE Transactions on Geoscience and Remote Sensing,* Vol. 29, No. 3, May 1991.

[17] M.R. Wohlers, S.Hsiao, J. Mendelsohn, and G. Gerdner, "Computer Simulation of Synthetic Aperture Radar Images of Three-Dimensional Objects," *IEEE Transactions on Aerospace and Electronic Systems,* Vol. AES-16, No. 3, May 1980.

Image Understanding

[Castleman] Castleman, K., *Digital Image Processing,* Prentice-Hall, 1979.

[Maragos] Maragos, P., “Tutorial on advances in morphological image processing and analysis,” *Optical Engineering,* vol. 26, no. 7, pp. 623-632, July 1987.

[Parker 94] Parker, J., *Practical Computer Vision Using C,* Wiley, 1994.

[Parker 97] Parker, J., *Algorithms For Image Processing And Computer Vision,* Wiley Computer Publishing, 1997.

[Unser] Unser, M., "Sum and Difference Histograms for Texture Classification," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vol. PAMI-8, 1:118-125, 1986

[Weeks] Weeks, A., *Fundamentals of Electronic Image Processing,* SPIE/IEEE series on imaging science & engineering, 1996.

Fourier Transform

[Duhamel90] Duhamel and Vetterli, "Fast Fourier transforms: a Tutorial Review and State of the Art," *Signal Processing*, vol. 19, pp.259-299, April 1990.

[Cooley] Cooley and Tukey, "An Algorithm for Machine Computation of Complex Fourier Series,," *Math. Comp.*, vol. 19, pp.297-301, April 1965.

[Ganapa] Ganapathiraju, Hamaker, Picone and Skjellum, "Analysis and Characterization of Fast Fourier Transform Algorithms," MS State High Performance Computing Laboratory, Oct. 1997.

[Duhamel84] Duhamel and Hoolomann, "Split Radix FFT Algorithm," *Electronic Letters*, vol. 20, pp.14-16, Jan 1984.

[Rader] Rader, "Discrete Fourier Transforms when the Number of Data Samples is Prime," *Proc. of the IEEE*, vol. 56, pp.1107-1108, June 1968.

[Frigo] Frigo and Johnson, The FFTW web page, http://theory.lcs.mit.edu/~fftw

[Frigo99] Frigo, "A Fast Fourier Transform Compiler," MIT Laboratory for Computer Science, Feb. 16, 1999.

[Saidi] Saidi, "Decimation-In-Time-Frequency FFT Algorithm," *Proc. of International Conference on Acoustics, Speech, and Signal Processing*, vol. III, pp.453-456, Adelaide, Australia, April 1994.

[Johnson] Frigo and Johnson, The BenchFFT web page, http://theory.lcs.mit.edu/~benchfft

Data Management

[Guttman] Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOID*, pp. 47-57, June 1984.

[Kornacker] Kornacker, Banks, "High-Concurrency Locking in R-Trees," *Proceedings of 21st International Conference on Very Large Data Bases,* pp. 134-145, September 1995.

[Sellis] Sellis , Roussopoulos, and Faloutsos, "The R+-Tree: A Dynamic Index for Multi-Dimensional Objects," *Proc. 13th International Conference on Very Large Data Bases*, pp. 507-518, Brighton, September 1987

Appendix A: DIS Benchmark Suite: Data Management Software Design

This document describes the baseline software design of the Data Management benchmark for the DIS Benchmark Suite. The document is separated into two parts; the first part presents the design approach at a high-level, describes various implementation decisions, and explains the interaction between the individual function descriptions. The second part presents the low-level descriptions of the individual routines and methods which make up the baseline application.

A. DESIGN DESCRIPTION

A.1 GOALS

Several goals were established and followed for this design:

- Accurately follow the Data Management specification as described in the DIS Benchmark Suite[AAEC-1].
- Strike a balance between easily understandable source code and optimized source code. Optimization usually has the effect of making software implementation more complex or more difficult to immediately understand. However, the baseline design should represent a "reasonable" database application and the baseline performance figures should be comparable to a real application which indicates some optimization. The choice of whether to use a specific optimization technique is determined by the amount of performance increase versus the degradation of the general user to determine the underlying process.
- Provide a complete application which is reasonably robust for a general execution. The application should handle generic errors (unable to locate input files, memory allocation failures, etc.) gracefully, and return control to calling process without an abrupt failure. The application should not be expected to handle hardware specific or special errors unique to a platform or hardware configuration. Also, when errors would require a large amount of code to detect and/or fix, the design reverts to the primary goal of optimization/understandability.
- Provide baseline source code which is relatively easy to understand and modify to a particular implementation approach. The design should allow the alteration of the underlying database algorithms without causing a major shift in the design paradigm.
- Follow the DIS Benchmark C Style Guide[AAEC-2] for the development phase of the baseline source code. The style guide lists several aspects of the source code which can be incorporated into the design segment.

A.2 CONVENTIONS

Several conventions are used in this document to clarify the design and implementation of the benchmark.

- A function name will be *italicized* when referenced within a text setting.
- Structures are in **bold** type when referenced within a text setting.
- Control flow in diagrams is denoted by arrows on solid lines.

- Subroutines and/or actions within statements, denoted by rectangular boxes, are referenced by dashed lines.

- Subroutines are denoted by "bold" rectangular boxes.

A.3 DESIGN OVERVIEW

The Design Overview section consists of a review of the requirements given by the DIS Benchmark Specification [AAEC-1] for the Data Management Benchmark. This is followed by the high level design description for the application as a whole which includes control flow and a separation of the required tasks into modules. The modules and their respective routines are described in following sections.

A.3.1 Benchmark Specification Requirements

The Data Management benchmark consists of the implementation of a simplified database which uses the R-Tree indexing algorithm. The R-Tree index is a height-balanced containment structure which uses multidimensional hyper-cubes as keys. The intermediate nodes are built up by grouping all of the hyper-cubes at the lower level. The grouping hyper-cube of an intermediate node completely encloses all of the lower hyper-cubes, which may be points. The application must respond to a set of command operations: *Initialization*, *Insert*, *Delete*, and *Query*, queries being either key-based or content-based. The commands are issued to the system in a batch form as a data set.

The *Initialization* command operation specifies the fan or order of the index tree.

The *Insert* command operation places a new data object into the database with the specified attribute values. Each *Insert* command contains all the information contained by the data object, including the hyper-cube key and list of non-key attribute values.

The *Query* command operation searches the database and returns all data objects which are consistent with a list of input data attribute values. The input attribute values can specify attribute values which are key, non-key, or both. A data object is defined to be, or is considered consistent with, the *Query* when the input values intersect the stored values of the data object.

The *Delete* command operation removes all objects from the database which are consistent with a list of input data attribute values. The types and conditions of the input attribute list, as well as the description for consistency, are the same as for the *Query* operation.

Each entry in the database will be specified by the *Insert* command and will be one of three types: Small, Medium, or Large. Each data object has a set of attributes which consists of the key and non-key attributes. The first eight attributes represent the index key and is specified as eight 32-bit IEEE floating-point numbers representing a four-dimensional point in Euclidean space denoted as the T-position, X-position, Y-position, and Z-position, of both the "lower" and "upper" points of a hyper-cube, respectively. Also, each object has a constant number of non-key attributes or parts. A non-key attribute is an 8-bit NULL-terminated ASCII character sequence of maximum length 1024. The number of attributes assigned to each data object type is given in Table A.1. The attributes for a given data object reference each other as a single-linked list and the data object holds a reference to the first attribute in the list which is defined as the head. Additional indices for use with non-key attributes are not permitted for this benchmark. Thus, a

Query operation which contains no key search information will search the entire database for consistent entries.

Table A.1: Data Object Types

Object Type	Code	No. of Non-Key Attributes
Small	1	10
Medium	2	17
Large	3	43

A.3.2 Design Description

The benchmark requires a complete application which can read the input file and process all of the commands. Figure A-1 shows a high-level execution flow for a generic benchmark database application. The first step is an initialization step which would create the index, open files, etc. The application then enters a loop which gets the next command code, collects metric information required by the specification, and then the execution branches based on the code. Each branch gets the appropriate command from the input and applies the command to the index. The "middle" branch is for the Query command and has an extra step of delivering the response of the Query to the output. The loop is repeated until there are no more commands to process. After the loop, an exit step is taken to close files, free memory, etc.

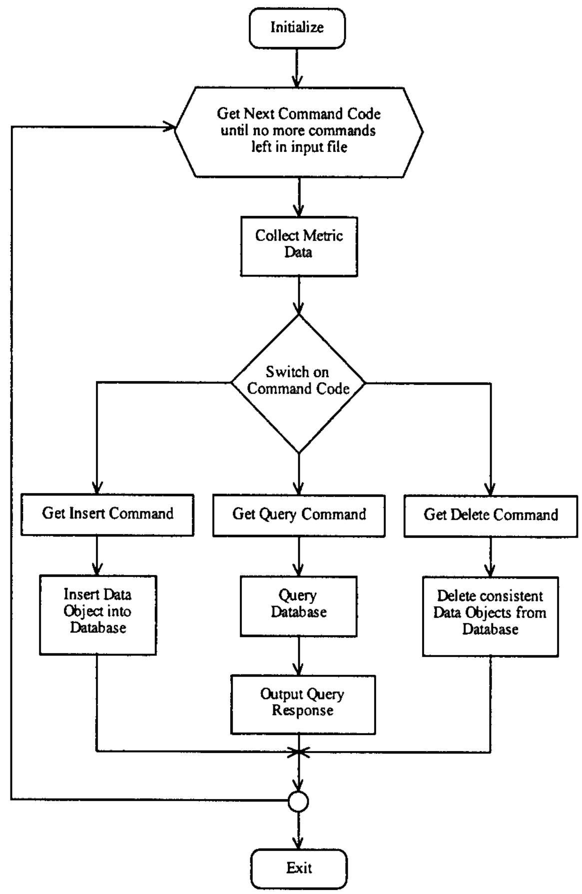


Figure A-1: Application Execution Flow

The outline above indicates that some of the tasks will require input (getting the command code and commands) and output (query response output and metric data output) functionality. The index management requires three separate tasks for each command. Finally, the metric data collection is another task required by the specification. The baseline design separates the application tasks into three separate modules: Database, Input & Output, and Metrics. The Database module implements the R-Tree index which includes index maintenance and query response. The Input & Output module handles the input of the data set, output of the query responses from the database module, and the output of the metric performance figures. The Metrics module determines the timing information for the baseline performance and any other statistical measures applied to the baseline application. The modules are used to allow separate development of portions of the baseline application simultaneously. Also, the different benchmark implementors can pick and choose which module, if any, to develop using their own implementation. Complete separation of tasks into their modules would normally require a layer of interface routines which would be used to transfer data between the modules. This extra layer of interface would increase the size of the baseline source code and introduce complexity for a relatively simple application. In order to minimize the source code size and complexity, the extra layer of interface will either not be used or will be incorporated into some of the routines in the modules. This "incorporation" will cause some of the routines to be specific to other modules besides their own, i.e., the modules will not be completely "plug and play". In order to minimize the effect on all of the modules, the Input & Output module is chosen to contain all of the effects, i.e., the Database and Metrics modules will not contain any information outside of their respective modules, but the Input & Output will.

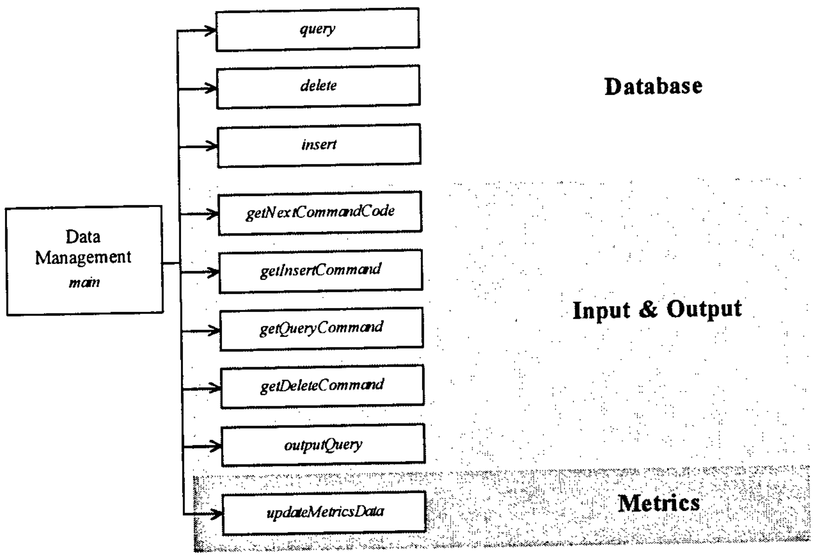


Figure A-2: Modules and High-level Routine Hierarchy

A functional hierarchy for the baseline design is shown in Figure A-2. Each function is represented by a box and is connected in the hierarchy by a solid line representing the calling function. Also, the functions are grouped by module and the modules are delimited by shaded boxes. The only function which does not have a calling function is the *main* function which is called by the user or system. The expansion of Figure A-1 into more detail is straightforward and is shown in Figure A-3. The figure shows an expanded series of steps and a set of routines which support each step.

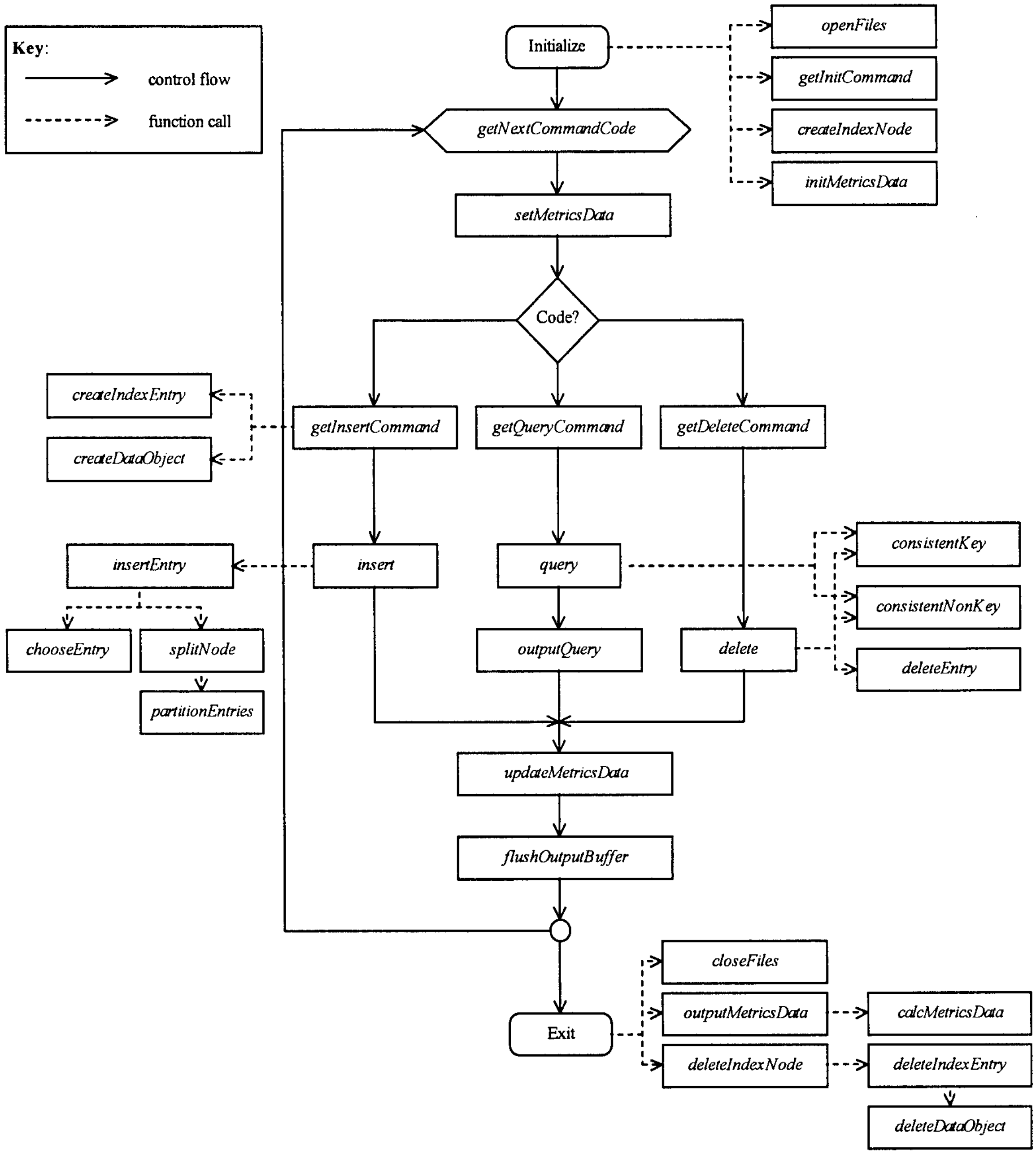


Figure A-3: Detailed Application Execution Flow

The Initialize step is used to initialize all three modules. The Input & Output module is initialized by opening the appropriate files for input of the data set and output of the query responses and metrics. The Database module is initialized by two routines. The first gets the initialization command from the input dataset. This command specifies the fan or order of the index and is always first within the file and never repeated. The second initialization routine is *createIndexNode* which creates the root node of the new index. The Metrics module is initialized by its own routine which will access system level timing routines to determine the start time of the application.

The step where the Metrics module collects data is now split into two steps and another step is added between them which flushes an output buffer. The secondary metrics for the benchmark requires timing of the individual commands. The time for a non-response command operation, insert and delete, to complete is defined as the time difference between the time immediately before the command is placed in the database input queue and the time immediately before the next command is placed in the same input queue. The time for a query command operation to complete is defined as the time difference between the time immediately before the command is placed in the input queue to the time immediately after the response is placed in the output queue. The splitting of the metric collection step allows the routine to satisfy these metric definitions.

The branch of the flow which processes an insert command contains two steps. The first, *getInsertCommand*, gets the command from the input and delivers a data object for insertion by the *insert* routine. This requires that the *getInsertCommand* routine have knowledge and access to a routine within the Database module to create a data object. The second step of the branch is the actual insert into the index. The tasks required of an insert are detailed by the specification and the routines shown in Figure A-3 directly correspond to these tasks.

The branch of the flow which processes a query command contains three steps. The first, *getQueryCommand*, gets the command from the input and delivers an index key and non-key attribute list to be used by the *query* routine. Note that "missing" values in the input command (see specification) are defaulted to wild-card values and the *getQueryCommand* will insert the appropriate wild-card values into the index key for use by the *query* routine. This could be handled by the *query* routine, but it was felt that the specification of "missing" values was an attribute of the input format and thus should be handled by a routine belonging to the Input & Output module. The *query* routine should make use of two subroutines, *consistentKey* and *consistentNonKey*. These routines check the key and non-key values of the stored index data with the input search values. The final step of the query branch is the output of the query response. Because of the metric data collection considerations (see paragraph above), the routine *outputQuery* places the query response into an output buffer and not necessarily to the output file. The buffer is then flushed when appropriate.

The delete command is processed in two steps. The first, *getDeleteCommand*, gets the command from the input and delivers an index key and non-key attribute list to be used by the *delete* routine. Note that "missing" values in the input command (see specification) are defaulted to wild-card values and the *getDeleteCommand* will insert the appropriate wild-card values into the index key for use by the *delete* routine. This could be handled by the *delete* routine, but it was felt that the specification of "missing" values was an attribute of the input format and thus should be handled by a routine belonging to the Input & Output module. The *delete* routine will search the index in a manner similar to the *query* routine and should use the same routines for checking, *consistentKey* and *consistentNonKey*. Also, the *delete* routine uses a recursive routine to descend the index removing appropriate data objects and ascending the index removing empty nodes.

The Exit step is used to exit each module. The Input & Output module will close the appropriate files and output the metric data. The output of the metric data will exit the Metrics module by calculating the metric statistics. The Database module will free the memory of the index which deletes all nodes, entries, and data objects which have been placed in the index during the execution.

A.3.3 Error Handling

One of the goals for the Data Management software design is to produce a robust application which will gracefully handle most errors. The strategy by which these errors are handled is described in this section and is applied uniformly in the application. The primary approach consists of the return of integer codes from each routine which can fail. A routine will have a successful return code, indicating that the specific task required of the routine was accomplished, or one or more error return codes which indicate the specific error that occurred. The return code indicates the state of the process and any other output data and not necessarily that no error occurred during the execution of the routine. For example, if an error occurs during a subroutine but the subroutine recovers, a successful code is returned to the calling process indicating that all output data and the current execution thread can proceed normally. This approach implies several characteristics for each routine:

- Each routine will handle all "local" errors. A local error is one caused directly by the system, e.g., opening files, reading/writing to/from a stream, etc., or by calls to other functional subroutines. This does not indicate that a routine will abort the execution thread, rather the routine will return an appropriate error code to the calling function.

- Each routine will "clean-up" before the return. If an error occurred, any memory allocated during the execution of the routine will be unallocated.

The only exceptions to the return code approach for the baseline design is for system routines which prescribe a different method and for baseline routines which closely mimic these system calls. The best examples are the memory allocation functions which return a pointer to the allocated memory or NULL if an error occurred. Each routine which can fail, lists the success and error return codes as part of the function description.

All error and/or unusual conditions which arise during execution of the application will have a descriptive comment placed in the error stream preceded by the names of the routines where the condition occurred, i.e., an error occurring within subroutine *B* which was called by subroutine *A* would have the message,

A> B> error message

where each subroutine name and the actual message are separated by ">". The message system is accomplished by two routines: *errorMessage* and *flushErrorMessage*. A local buffer is kept by the routines to allow storage of the message and routine names before flushing and the size of the buffer is set such that exceeding the limit is extremely rare . The extreme case when the prepended message is larger than the buffer size will cause the *errorMessage* routine to immediately flush the current error buffer along with a message indicating the premature flush. The *errorMessage* routine inputs two parameters: (1) A character string which should either contain a text representation of the condition that occurred or the name of a routine which should be prepended to the current message, (2) An boolean value indicating that the first parameter message should replace the current error buffer contents, or the first parameter message should be prepended to the current error buffer contents. The *flushErrorMessage* routine takes no parameters as input and simply places the current contents of the error buffer into the standard error stream. A call to the *flushErrorMessage* does not clear the buffer contents.

A.3.4 <u>Testing</u>

The baseline source code and application requires testing of the software at the unit and system level. The unit testing description for each routine described in Part II of this document is included with the routine description. Most follow a similar pattern where a known input/output data pair is used for the testing. The input portion is provided to the routine and the output of the routine is checked with the output portion of the pair. If the two sets of output match, the routine is considered successfully unit tested. Another test consists of memory analysis to attempt to discover memory leaks and similar errors. If a specific routine requires further unit testing, the unit testing method is given with the routine description.

A.4 DATABASE

The database is specified to respond to three commands: Insert, Query, and Delete. This section describes the Database module. This description includes how the R-Tree index is implemented and how the three required commands are applied to the index.

A.4.1 <u>R-Tree Index Requirements</u>

The requirements for the R-Tree index specified by the DIS Benchmark Suite are the same as those defined for a generic R-Tree index. A general R-Tree has the following properties:

1. All leaves are at the same level (height-balanced).

2. Every node contains between kM and M index entries unless it is the root. (M is the order of the tree).

3. For each entry in an intermediate node, the sub-tree rooted at the node contains a hyper-cube if and only if the hyper-cube is "covered" by the node, i.e., containment.

4. The root has at least two children, unless it is a leaf.

The data management benchmark requires the R-Tree structure be maintained during execution, but the particular methods used to maintain the R-Tree are left for individual implementors. The baseline design will generally follow the R-Tree algorithm as described by Guttman[Guttman] with any exceptions noted in the routine descriptions.

A.4.2 <u>R-Tree Index Implementation</u>

A brief description of the index implementation is provided in this section which will help clarify later discussion of the routines which manipulate or use the index. The index is implemented as a set of data structures of three basic types as shown in Figure A-4.

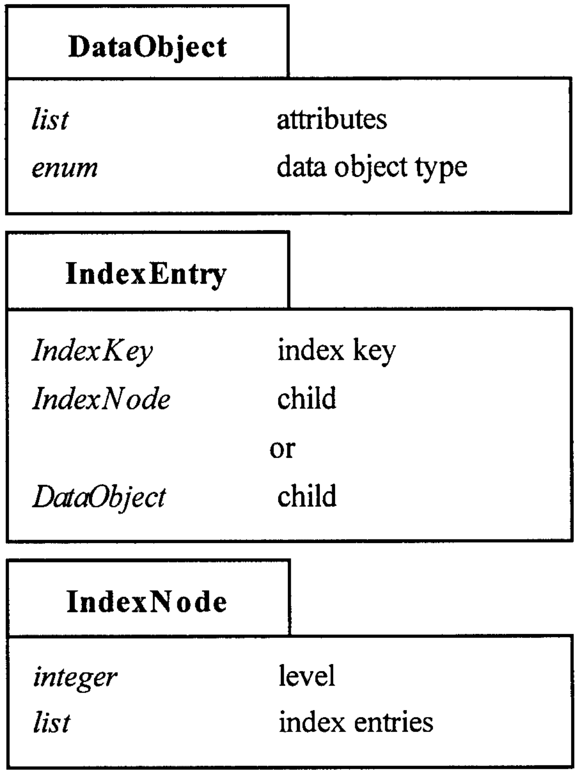


Figure A-4: Basic R-Tree Structures

The first data structure shown is for a **DataObject** which is the basic storage structure of the database. A **DataObject** holds all of the object's attributes as a *list* of attributes some of which are key and the rest non-key. Also, an enumerator which stores the type of data object is assigned to the **DataObject** structure. The number of non-key attributes contained in the *list* is determined by the data object type enumerator and is given in Table A.1. The number of key attributes in the *list* is determined by the dimension of the R-Tree which is eight for the DIS Data Management benchmark.

The second data structure shown is for an **IndexEntry** structure which is used as a reference structure to other portions of the index. The **IndexEntry** contains two pieces of data. The first is an *IndexKey* which is the key information for the object that the **IndexEntry** references, i.e., if the **IndexEntry** references a **DataObject** then the *IndexKey* of the **IndexEntry** is the appropriate key values of the **DataObject**. The second piece of data in an **IndexEntry** structure is a reference to either a **DataObject** structure or an **IndexNode** structure. The **IndexEntry** child member will reference a **DataObject** when the **IndexEntry** is located at the leaf level, otherwise, the child member will reference an **IndexNode** structure, i.e., a branch of the index. The value of the *IndexKey* member of the **IndexEntry** structure when the **IndexEntry** references an **IndexNode** is the enclosing hyper-cube which minimally contains all of the separate *IndexKey* structures associated with that **IndexNode**. This is the containment characteristic of the R-Tree index.

The third data structure shown is for an **IndexNode** structure which is used to contain a *list* of **IndexEntry** structures. The number of **IndexEntry** structures in the *list* can vary between one and the fan or order of the index. Also, the **IndexNode** structure contains an *integer* value which specifies the level at which the node resides within the index where the leaf is defined to be level zero and the level increases as the tree is ascended, i.e., the root level is always greater than or equal to the leaf level.

Figure A-5 shows a schematic of the index tree as described in the previous paragraphs. The **IndexNode** structures contain a set or list of **IndexEntry** structures. Each **IndexEntry** structure references either a **DataObject** or an **IndexNode** depending on the level where the **IndexEntry** is placed. Note that one of the requirements for an R-Tree is that the index is always balanced, i.e., the distance (number of levels) between the root and a leaf node is the same for each leaf. This implies that if a node is a leaf, all of the node's siblings are also leaves.

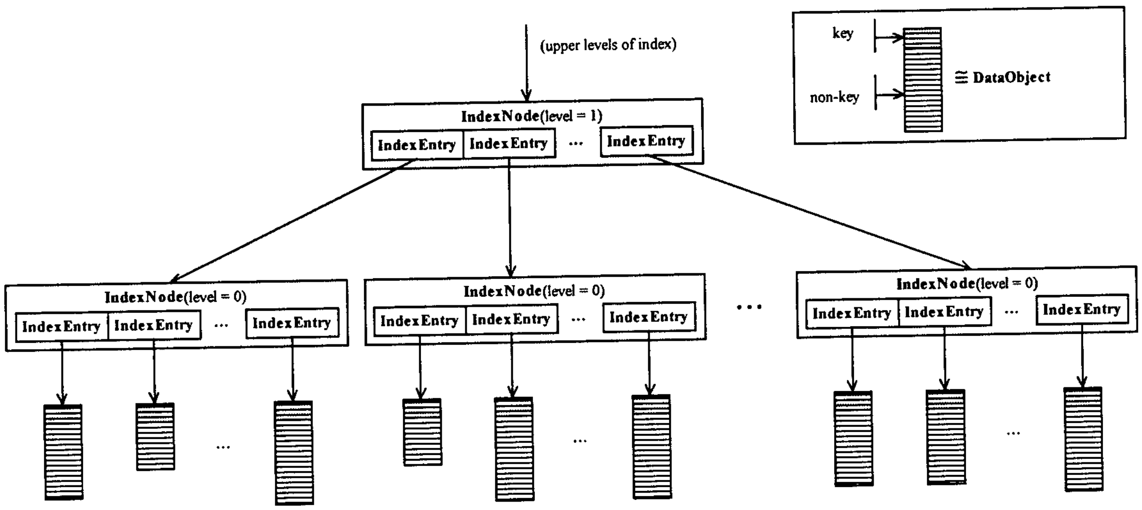


Figure A-5: R-Tree Index Schematic

The index is manipulated through three main routines: *insert*, *query*, and *delete*. The routines were chosen to represent each command required by the database. Each of these routines is described in the following three sections. The description contains background on the tasks performed by the function, notes on any specific design decisions which would have particular impact on anyone modifying the design for hardware considerations, a function hierarchy, and control flow diagrams.

A.4.3 <u>Insert</u>

The Insert command places a new data object into the index. The insert command also specifies the index order or fan (which allows the input dataset to specify the fan at run time via the *Initialization* command). A new index entry is created and assigned for the data object. The insert method descends the tree until the leaf level is reached. Note that the leaf level is zero and the level increases as the tree ascends, i.e., the root level is always greater than or equal to the leaf level. The branch or node chosen for descent is determined by comparing the penalty for all possible branches and the new entry. The branch which has the smallest or minimum penalty is chosen. Once the specified level is reached, a node is chosen on that level which yields the minimum penalty and the new index entry is placed on that node. Placement of the entry onto the node may exceed the specified fan which causes the node to split. The node split separates the union of the old entries of the node and the new entry into two groups. One group is placed back onto the old node, and the other group is placed on a new node created for that purpose. The new node is then placed onto the parent of the old node. This may cause the parent to split and an identical splitting process is carried out on the parent, which may cause its parent to split, etc. This splitting may ascend to the root node, which by definition has no parent and so is a special case. When the root node is split, a new root is created which "grows" the tree. The old root and the node split off of the old root are then placed onto the new root, and the new root is returned as the updated index.

A comprehensive example of an index entry insertion, node splitting, and root node splitting with tree "growth" will help clarify the description above. The insertion of a new index entry into a current index is illustrated in Figure A-6. The index for this example has a fan of three, a current height of one, and indexes a total of nine data objects. Figure A-6 shows the example index and the new entry for insertion. The far right node is chosen for the leaf insertion which indicates that the chosen node has the smallest penalty of the three branches checked.

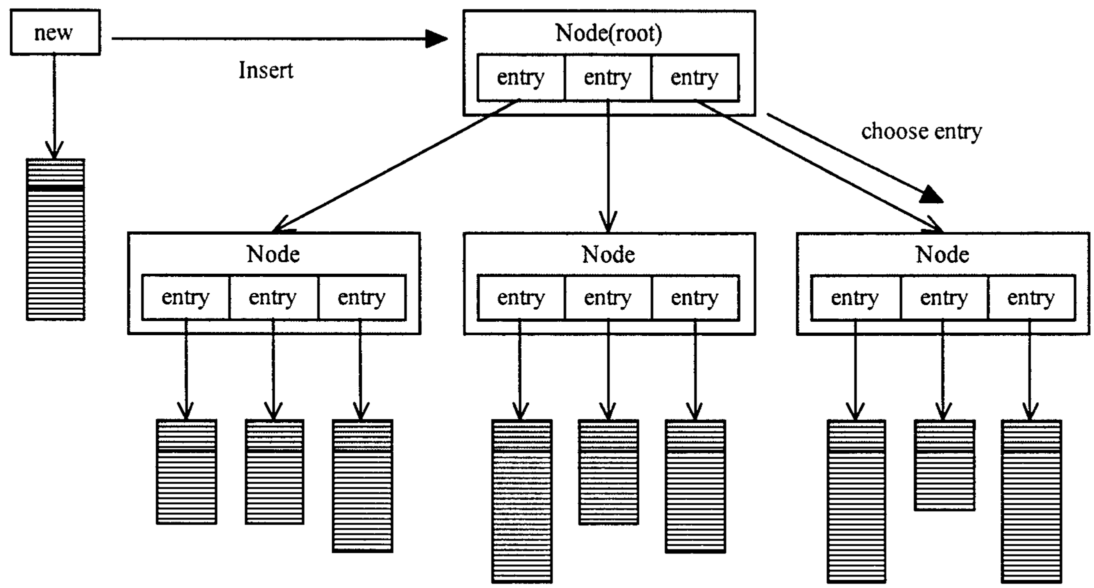


Figure A-6: Insertion of Entry into Index

The chosen node is split since it was full and there was no place for the new entry. A new node is created along with a new entry called split. The split entry/node is then placed on the parent of the chosen node, which happens to be the root node for this example. Figure A-7 shows the split entry/node and the placement of the split entry onto the parent. Note that for the split, four entries are divided between the chosen and new nodes, and the new entry is placed on the chosen node rather than the new or split node. The separation of the entries into two groups is called partitioning.

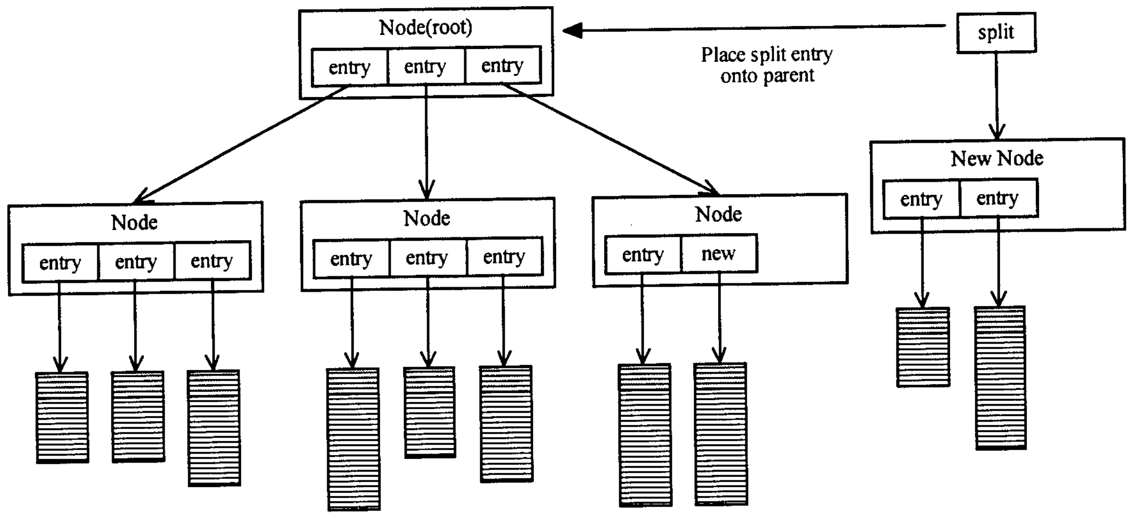


Figure A-7: Split of Leaf Node

The placement of the split entry onto the parent or root causes the root to split since the root is full. The split creates a new node/entry pair which is placed onto a newly created root node along with the old root. This "grows" the tree by one level. Figure A-8 displays the updated index after the insertion and root split. Note that the tree remains balanced where all of the leaves are on the same level. The final index has a current height of two and references a total of ten data objects.

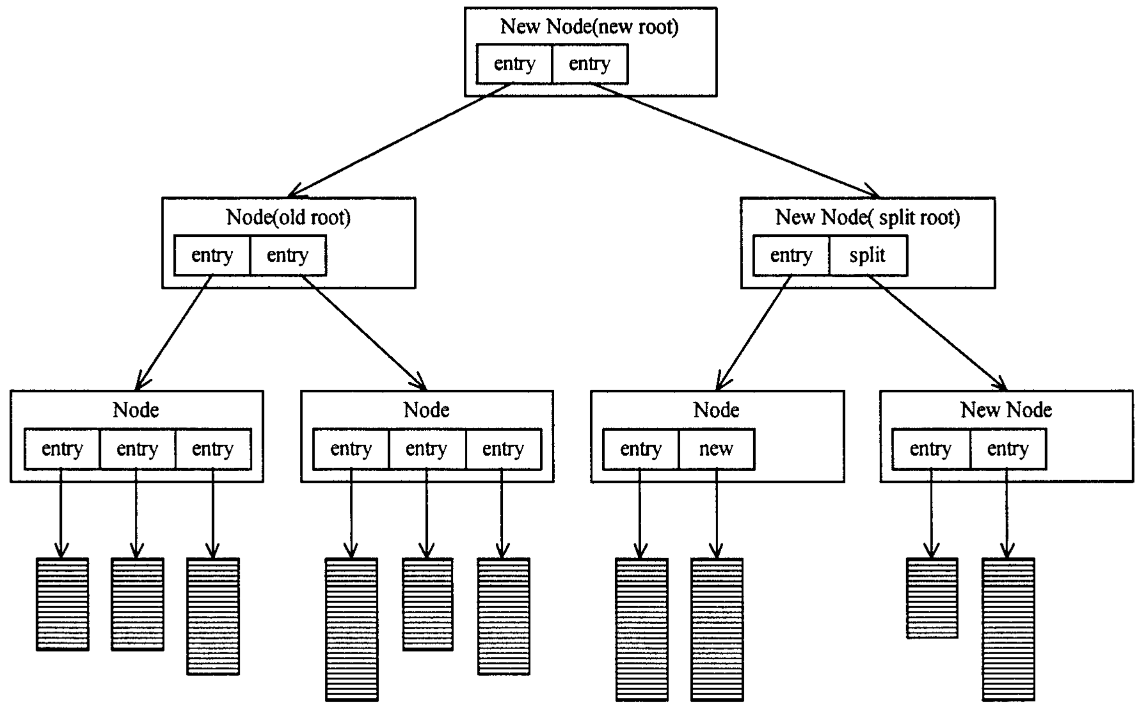


Figure A-8: New Index after Insert Command

The insert command is logically split into several different routines which represent each of the main tasks described above and a functional hierarchy is shown in Figure A-9. Also, utility routines for the creation and deletion of index entry and index nodes as well as penalty calculations and index key union routines are shown in the figure.

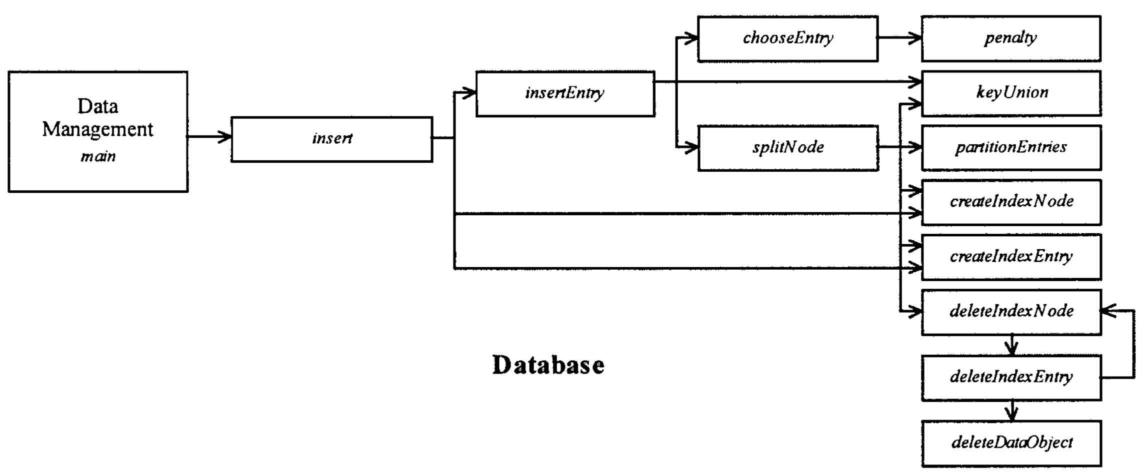


Figure A-9: Insert Command Function Hierarchy

The *insert* routine places a new index entry into the index using the recursive *insertEntry* subroutine. The input root node is used as the start of the tree descent. If the root was split, the tree "grows" by creating a new root and placing the old root and the split entry onto the new root and returning the new root as the updated index. A control flow diagram for the *insert* routine is shown in Figure A-10.

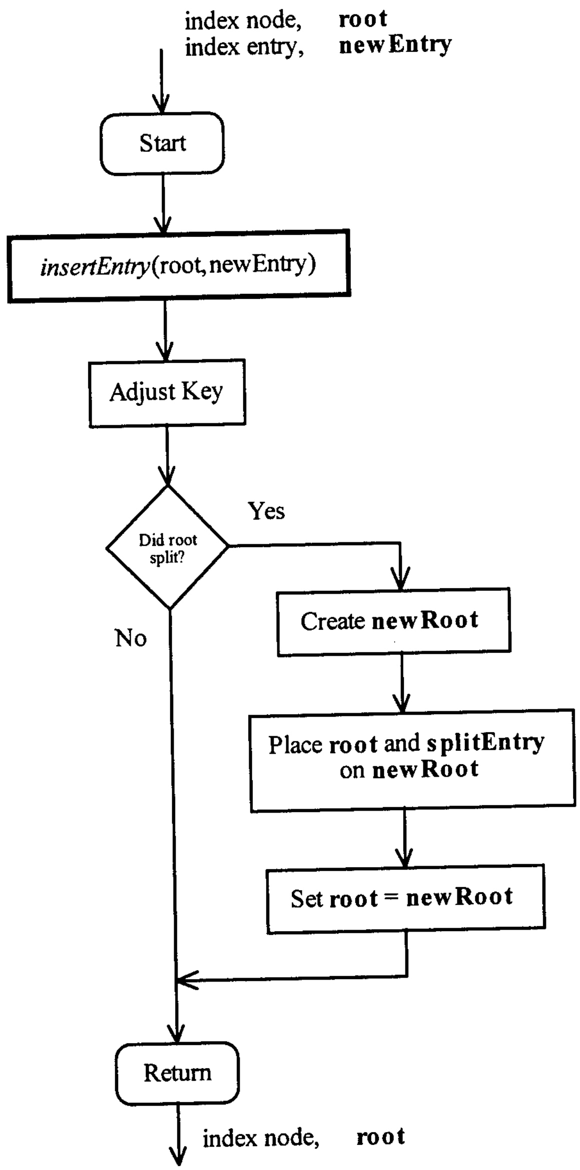


Figure A-10: Insert Control Flow Diagram

The primary reason that a non-recursive *insert* routine was chosen as a wrapper for a recursive *insertEntry* routine was for the special case of root node splitting. For every node in the index which splits, the parent node is responsible for placement of the new sibling split node/entry. This approach is valid for all nodes in the index except the root node which has no parent. Instead, the root node must "grow" the index. This unique task for the root node is best handled in a separate non-recursive routine and the recursive nature of all other insert/splits should be handled separately. A modified *insertEntry* which checks for root splitting is possible and would have the advantage of eliminating an "extra" subroutine since there would be no need for *insert*. However, the *insert* routine is used for this implementation because the modified *insertEntry* would check each node for root splitting, which isn't necessary, and the additional clarity of placing the special case of root splitting into a separate routine. The *insertEntry* routine control flow is shown in Figure A-11.

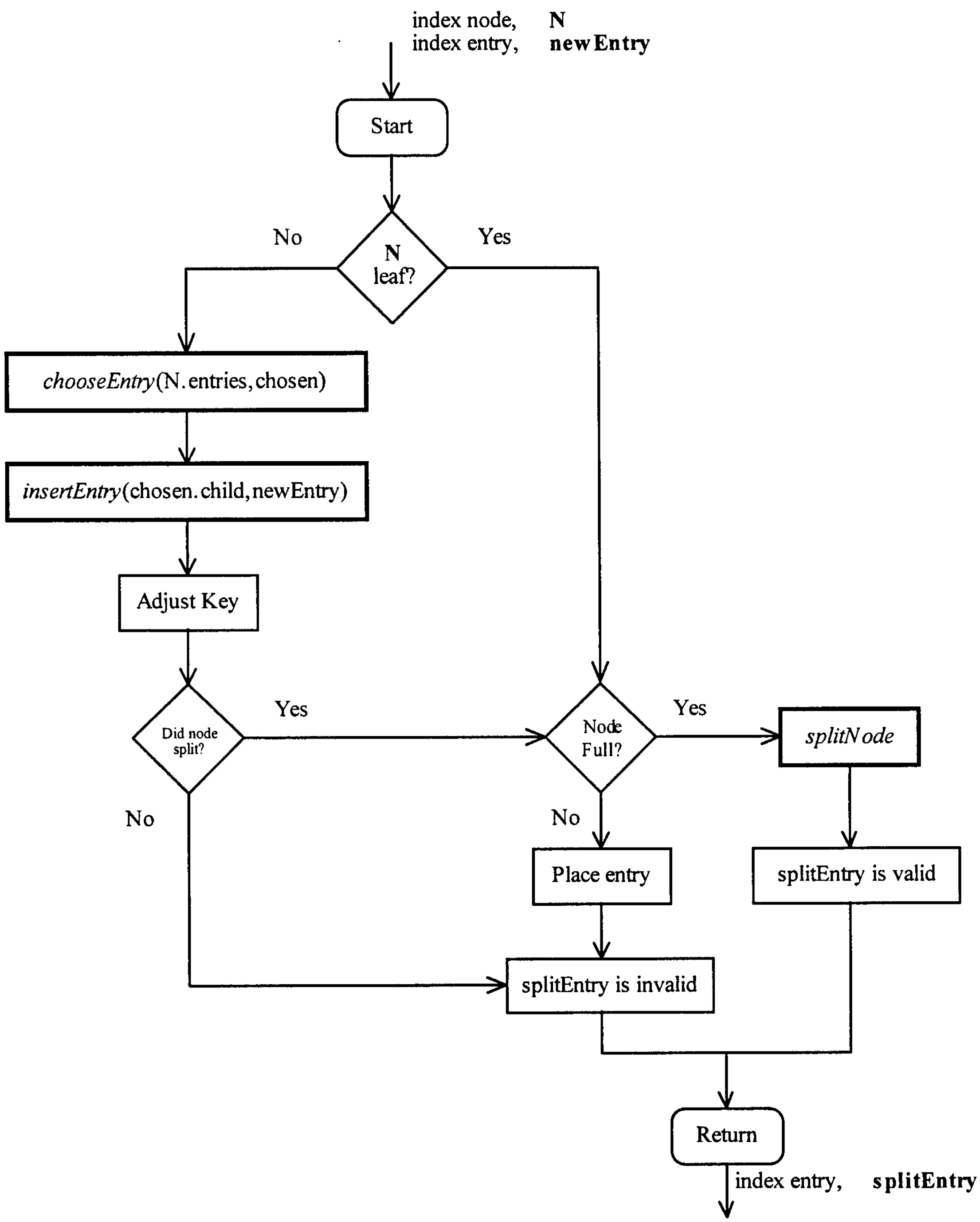


Figure A-11: Insert Entry Control Flow Diagram

Note that the *insertEntry* routine is recursive and is called for every node during the descent to the correct level. Also, the *insertEntry* routine calls two other routines called *chooseEntry* and *splitNode*.

The *chooseEntry* function is used to determine which branch of the current node to descend. The routine does this by determining which branch will yield the minimum penalty for the insertion. The penalty for the R-Tree index is the increase in hyper-cube volume.

The *splitNode* routine splits an input node into two nodes and divides up the entries via the *partitionEntries* function. The *partitionEntries* function is one of the most important routines implemented in the baseline source code. The *partitionEntries* function separates an input list of index entries into two groups. The method used for partitioning the entries is extremely implementation dependent. The basic idea is to set-up the two output index entry lists to have minimal bounding hyper-cubes which will improve later queries on the index since fewer branches of the index will need to be traversed to satisfy the query command. However, the method itself is probably the most computationally expensive of the insertion subroutines, because multiple loops through the index entry lists and penalty calculations are required for true "minimal" bounding hyper-cubes to be determined. If multiple branch searches is not prohibitive, i.e., a parallel search is possible or query response is not time consuming relative to an insert operation, then the partition subroutine can use a sub-minimal approach. The basic idea is not to spend too much time arranging the index when the pay-off, query performance, will not yield significant performance improvement. For example, if an insert command requires 10 seconds to minimally arrange the current index, and the insert is followed by 5 queries each of which requires 1 second to complete, then the total time of operation is 15 seconds. However, if an insert command would only require 5 seconds to sub-minimally arrange the current index, and each of the five queries can be accomplished in 1.25 seconds, then the total time of operation is 11.25 seconds which is an overall savings of 3.75 seconds even though the insert operation was "sub-optimal". In fact, the partition can simply split the input list into two equal groups ignoring the bounding hyper-cubes completely. The effect will be to cause new traversals of the index to descend multiple branches, but this trade-off may be acceptable for a given implementation.

A two-dimensional example of how the *partitionEntries* function operates and a modification will illustrate this point. The example consists of five entries to partition into two groups as show in Figure A-12. Objects 1 and 5 produce the "worst" or largest bounding index key so they are chosen as the first entries for the two new groups.

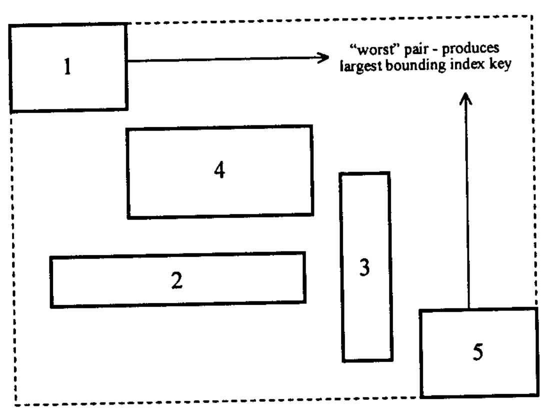


Figure A-12: Partition Entries Example 1

The index keys of all of the following Objects 2, 3, and 4 are done in a similar fashion and are shown in Figure A-13-Figure A-15. For each object 2-4, the penalty is found between the object and objects 1 and 5 The object is assigned a group based on which penalty is smaller, where a smaller penalty indicates an "attraction" to the group, i.e., the addition of the candidate box would cause less of an increase in area for the group compared to the increase in area for the other group.

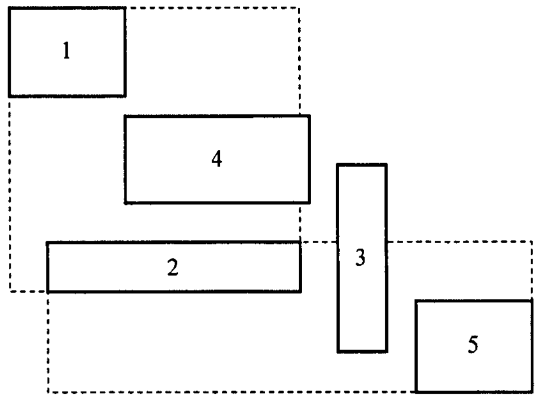


Figure A-13: Partition Entries Example 2

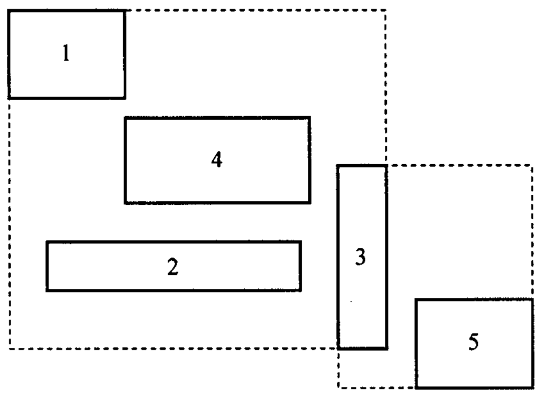


Figure A-14: Partition Entries Example 3

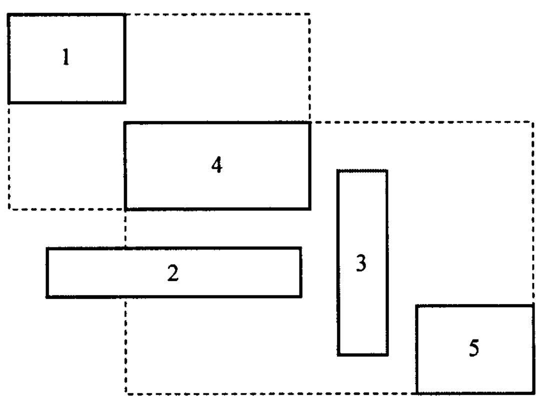


Figure A-15: Partition Entries Example 4

The five objects are now in two groups as shown in Figure A-16 and which produce minimal bounding index keys for each group. This partition will improve later index traversals since multiple branches need not checked since there is minimal (or in this case none) overlap of the stored index key entries. An example query key is shown in Figure A-16 which intersects (consistent) with Object 2. Note that the branch which contains Object 3 and Object 5 will not be searched for this query key.

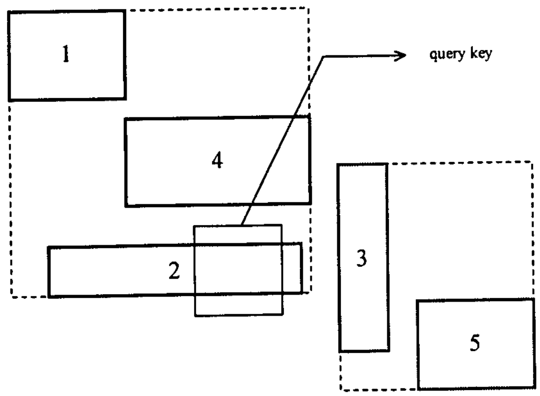


Figure A-16: Partition Entries Example 5

A very simple and computationally "cheap" method is to partition the input entries into two equal, or nearly so, groups. If this is applied to the previous example of Figure A-12, the following two groups are produced in Figure A-17:

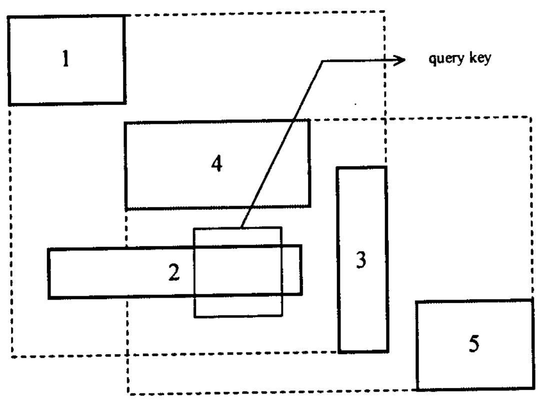


Figure A-17: Partition Entries Example 6

Object 1, 2, and 3 are assigned to the first group and Object 4 and 5 are assigned to the second group. This produces a very fast partition routine, but the same query key example for Figure A-16, which is also displayed in Figure A-17, will cause both branches to be searched, even though no objects will be found in the second group.

The utility routines which manage the creation and deletion of the index structures consist of three pairings: *createIndexEntry* and *deleteIndexEntry*, *createIndexNode* and *deleteIndexNode*, and *createDataObject* and *deleteIndexObject*. The routines allocates the correct amount of memory for each type of structure and sets default member values. The deletion routines will be recursively implemented, i.e., an index entry which is deleted will first delete either the index node or data object it references, and an index node which is deleted will first delete each entry that resides on that node, etc. Thus, an entire branch of the index can be deleted with a minimum of source code.

A.4.4 Query

The Query command inputs key and non-key information and returns all data objects currently in the index which are consistent with the query input. The Query command simply traverses the index and does not alter the index or data objects in any way. Since the query can operate on any index node regardless if it is the root or not, a recursive implementation is ideal. Since non-leaf entries reference index nodes rather than data objects, the query should treat leaf nodes and non-leaf nodes differently. Entries of non-leaf nodes which have consistent index keys compared with the input value, should query the child of the entry which is also an index node. Entries of leaf nodes are data objects and both the index key and non-key input values should be checked for consistency. Most of this processing can occur in a single routine with utility subroutines provided which will perform the consistency checks and check the validity of the keys before they are processed. The validity check is necessary to prevent a query search which would return nonsense values. Figure A-18 shows a functional hierarchy of the query command with the routines mentioned.

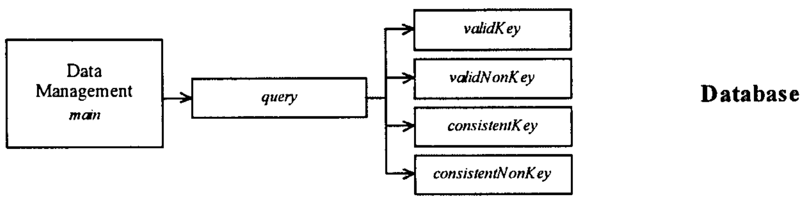


Figure A-18: Query Command Functional Hierarchy

Figure A-19 shows a flow chart of the query algorithm. The routine first uses the R-Tree index to find individual leaf nodes which point to data object's which have index keys which are consistent with the search key. The found data object's non-key attributes are then compared with the input list of non-key search values for consistency. The input list of non-key search values consist of the attribute code and a character sequence.

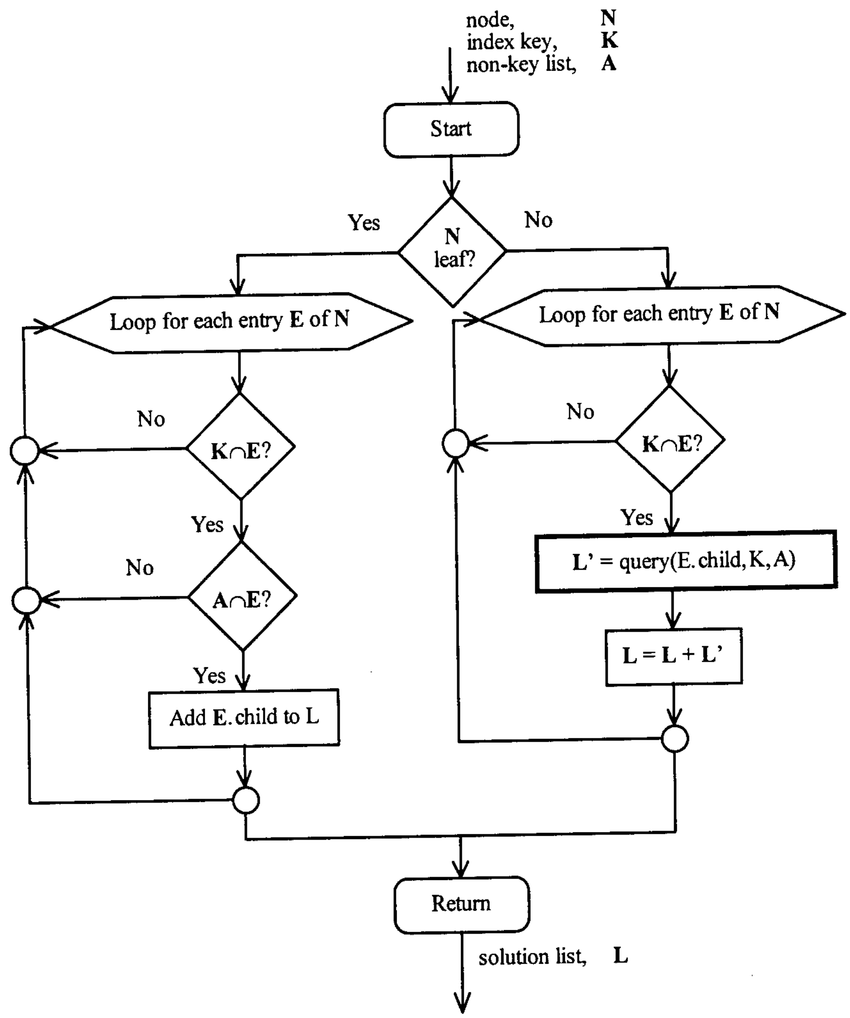


Figure A-19: Query Control Flow Diagram

The time required for a Query command is traditionally the primary metric for database applications. Most algorithms and implementations of the R-Tree index assume that faster query operation, even at the expense of index maintenance, is always desirable. This is not the case for the DIS Benchmark Suite[AAEC-1] which attempts to measure both the query performance and index maintenance performance via the Insert and Delete commands. The usual method of improving query performance is by minimizing the overlap of the index keys in the R-Tree, since an index with a relatively large amount of overlap will cause the query to descend multiple branches of the index tree. However, minimizing overlap is generally the most computationally expensive of the index maintenance methods. In traditional systems, this overhead is delegated to "low" user time or when the system is idle, but DIS assumes there will be no down time for index maintenance and that new systems will need to perform maintenance while still responding to user queries. Obviously, a trade-off occurs between query performance and index maintenance performance.

A.4.5 Delete

The Delete command removes one or more data objects from the index. The command inputs both key and non-key data and removes all data objects which are consistent with the input values. Thus, a portion of the command will resemble a Query by traversing the index checking key values. Once a data object has been found which is consistent with the input key and non-key search values, the data object and the index entry which references that object are removed from the index and the parent of the node where the deleted entry resides will need to adjust its index key since the child it references has changed. After one or more entries have been removed from the index for a particular node, some type of judgment must be made about the node and its remaining entries. If there are no entries left on the node, the node is no longer required and should be removed from the index. This requires that the entry which references the obsolete node be removed which may cause a chain reaction up the tree removing any nodes which are empty. The process could continue to the root node which would be removed if it had only one child, since any root node with only one child is redundant because the child could serve as a new root. If the node is not empty after the deletions, the number of entries left may be small compared to the fan or some pre-defined limit. A common practice is to specify a minimum fill factor and any node after the deletions whose number of entries is less than this factor would be removed and the remaining entries re-inserted into the index.

A comprehensive example of a data object deletion, node removal, and root node tree "shrinkage" will help clarify the description above. Figure A-20 shows an example index with a fan of three, current height of one, and indexes a total of ten data objects. Five data objects are to be deleted and are highlighted by dashed boxes. The process in choosing these objects would take the form similar to a query command, i.e., traversing the index using an input index key search value and choosing data objects by using the input index key and non-key search values.

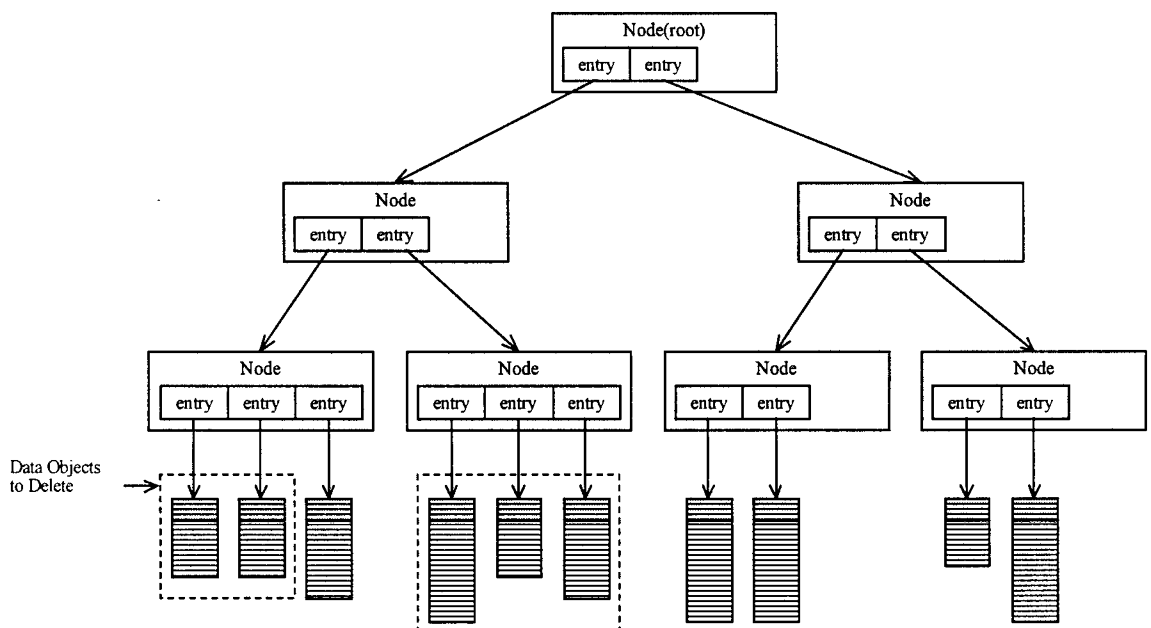


Figure A-20: Deletion of Data Objects from Index

Figure A-21 shows the index after the data objects have been removed. One node is completely empty and should be removed from the index. A sibling node has only one entry and will be considered "underfull" for this example. Note that removing the node requires that the entry be placed in a re-insertion list outside of the current index

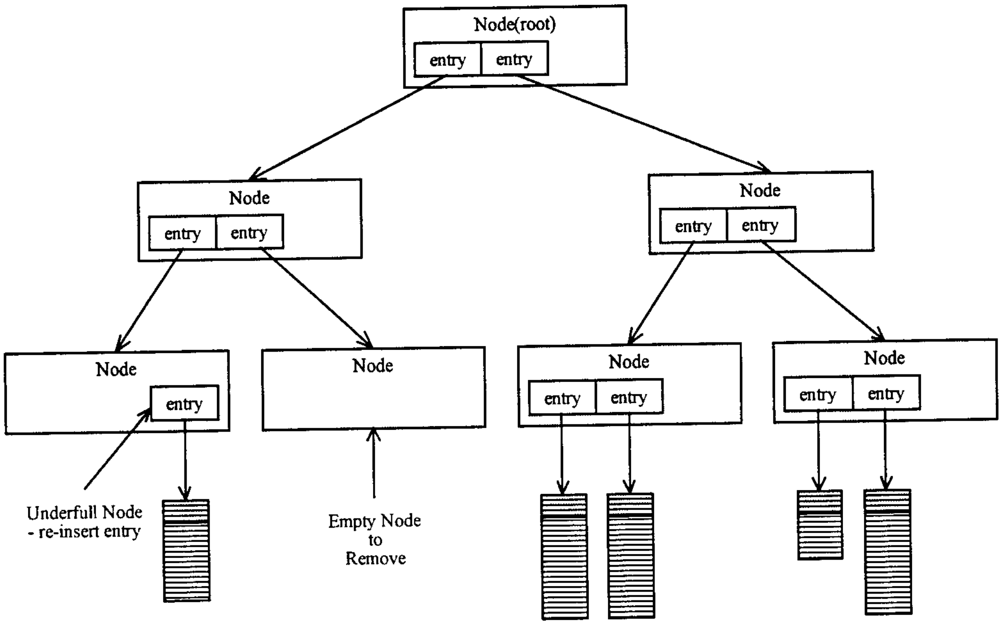


Figure A-21: Node Removal

Once the two nodes at the leaf level have been removed, a node one level higher is now empty and should be removed as well as shown in Figure A-22. This is part of the chain-reaction referred to in the delete command description above.

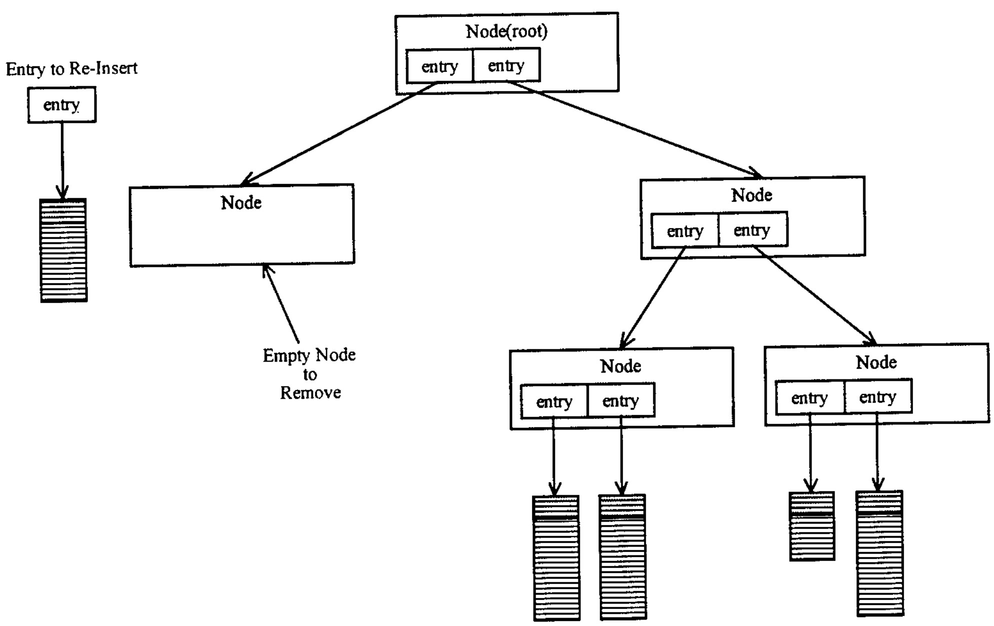


Figure A-22: Further Node Removal

Figure A-23 shows the index after the final node removal. The root now has only one child and violates the fourth condition for an R-Tree specification given in section A.4.1. The old root node is replaced by its child, since a root with only one child is redundant. Note that this process could be recursive, i.e., if the child of the old root also had only one child, then the "grandchild" of the old root would be the new root, etc., until fourth condition of section A.4.1 is satisfied.

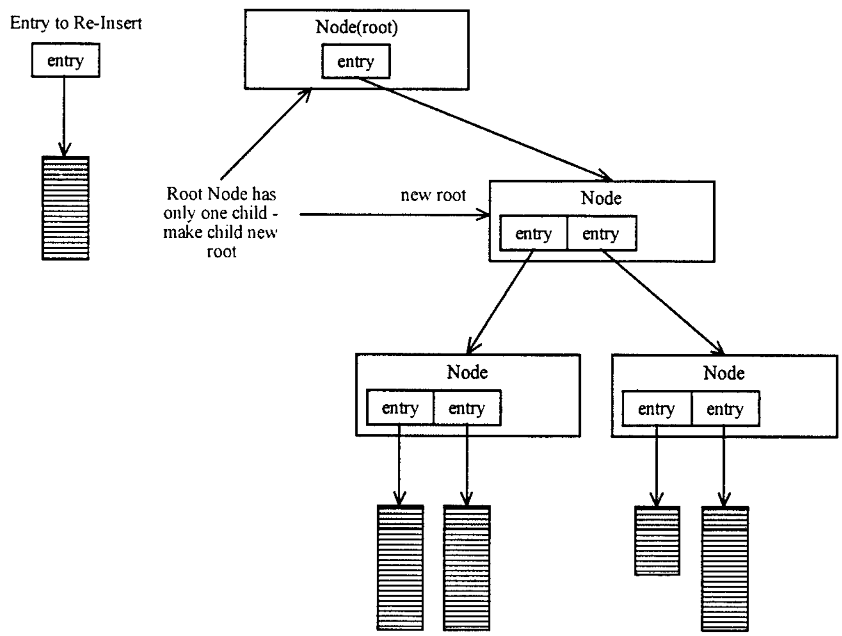


Figure A-23: Shrinking the Index Tree

The index after the node removals and root replacement is shown in Figure A-24. Any entries which were removed because their nodes were underfull are now re-inserted into the index. The example has only one entry to re-insert, although the length of the list is unlimited. For an entry which references a data object, the re-insert process should be identical to the process followed by the index for an insert command, i.e., tree descent by minimizing the penalty, possible node splitting requiring entry partition, and possible tree "growth". It is possible for an entry placed in the re-insert list to reference an index node rather than a data object. The entry for this case should reside at a level other than the leaf level and the insert process must be able to accept an index entry insertion with a specified level in order to keep the tree balanced, i.e., all of the leaf nodes are at the same level. The description of the insert process described in section A.4.3, particularly using the *insertEntry* routine, allows for a level to be specified.

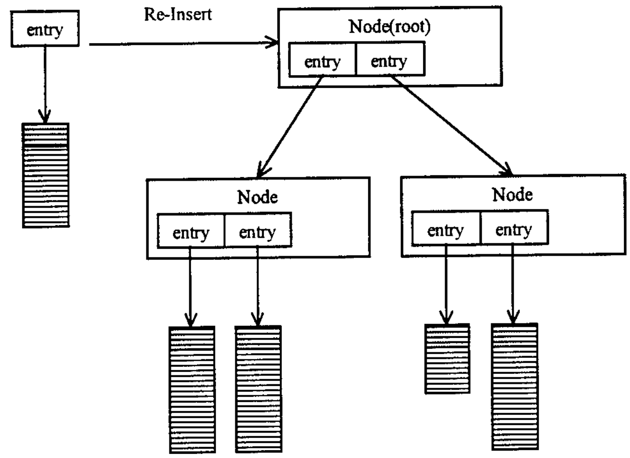


Figure A-24: Re-Insertion of Entry

The insertion of the one entry does not cause node splitting for this example and the final index after the complete delete command processing is shown in Figure A-25. The index has a current height of one and references five data objects using three nodes.

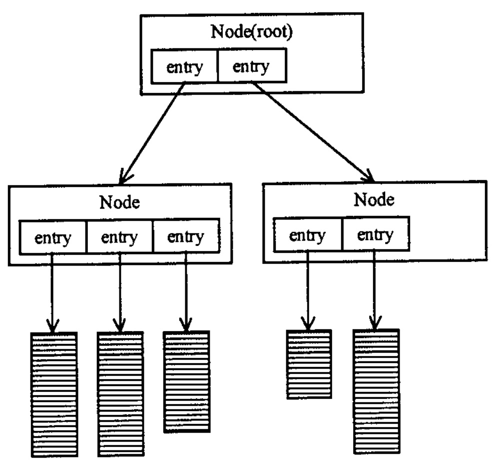


Figure A-25: New Index after Delete Command

There are two major issues for the delete command implementation. The first centers on when node removal occurs during the delete operation. Note that this issue is always present since the deletion of data objects can always cause an empty node to appear. The second issue deals with the concept of a node being "underfull" and is a design decision on whether to implement the behavior for index management.

Node removal should occur whenever an index node is empty. A node can become empty through data object deletion (leaf nodes) or a node being "underfull". The question of when removal occurs is the issue and can be implemented in one of two ways. The first approach would traverse the index removing the consistent data objects. During the traversal, the nodes are checked and removed immediately. After the index has been completely traversed, the delete command has been successfully processed and the index is fully updated requiring no further traversals of the index, although the root node may need to be replaced by a child as described earlier. The second approach would traverse the index removing the consistent data objects, but would not check the nodes for removal. Instead, after the traversal which removed the data objects, another traversal would take place which would "condense" the tree by checking each node for removal. The second approach has the primary advantage of separating the deletion behavior into two "steps" which can be independently implemented and executed. The idea would be that a condense routine would be implemented after a delete, but not necessarily "right" after. The tree could be condensed during idle time when no other commands are awaiting execution. However, the DIS Benchmark Suite specifically states that this type of idle condition will not be present during the execution of the baseline data. Also, the extra traversals which would be required would likely lead to an inefficient implementation for large indices. Since the benchmark specifies that index management performance is of primary concern, the first approach is chosen for the baseline design. Different architectures, especially parallel hardware, could make use of the condense routine approach. Although the system is always at a high load (commands are always awaiting processing), individual processors within a parallel architecture may experience idle time which could be taken advantage of by executing a condense routine.

The second design issue is the implementation of re-inserting entries of "underfull" nodes. The behavior is not required by the R-Tree algorithm which does not specify a minimum number of entries for an index node. However, the implementation of re-insertion allows the index to be further "condensed" which will improve performance for later index traversals and helps to avoid the undesirable condition of a "sparse" index where the tree has a large height for a relatively small number of data objects to reference. The introduction of this behavior significantly increases the complexity of the delete processing, since it introduces a new re-insertion index entry list which will also need to store the level that the entry needs to be inserted since entries other than leaf entries are possible. Also, an extra step is required during processing to account for the re-insertion process, and although most of the process would be identical to the insert command process, some slight modifications would be required and the extra level of processing would degrade the delete index management performance. The DIS Benchmark Suite specifically states that the data set will be highly dynamic with a relatively large number of index management commands, Insert and Delete, compared to Query operations. The baseline application will not implement re-insertion behavior to reduce the complexity of the baseline source code and will rely on the dynamic nature of the data indexed by the database to place the burden of efficient index traversal on the insertion operations.

A functional hierarchy for the delete process is shown in Figure A-26. Four of the routines are also used by the *query* routine to key and non-key consistency checks and to check key and non-key validity.

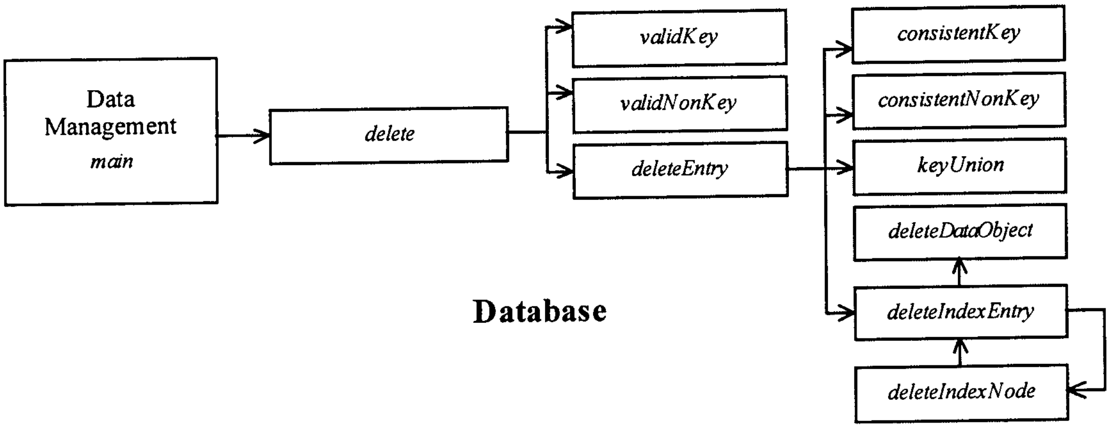


Figure A-26: Delete Command Function Hierarchy

Since node removal occurs immediately, the delete process for an individual node can be placed in a recursive routine called *deleteEntry* which is shown in Figure A-27. This routine will process leaf and non-leaf nodes differently. The entries for leaf nodes are checked for consistency with the input search values. If appropriate, the data object and entry are removed and a flag set to indicate to the parent of the entry to adjust the index key. The entries for non-leaf nodes are check for consistency with the input of the key search values only, since no non-key values are present to check. If the entry is consistent, the *deleteEntry* routine is recursively called to possibly remove data objects "beneath" the current node. Upon return from the recursive call, the index can be in one of three conditions: (1) No entries were removed, so no key adjustment is required, (2) One or more entries were removed but the child node is not empty, so only a key adjustment is required, and (3) One or more entries were removed and the child node is empty, so the node should be removed negating a key adjustment, although the parent of the current node will still require a key adjustment. Note that both "branches" of the control flow are similar to the Query command shown in Figure A-19, with the difference in what is done with the result of the key and non-key consistency checks.

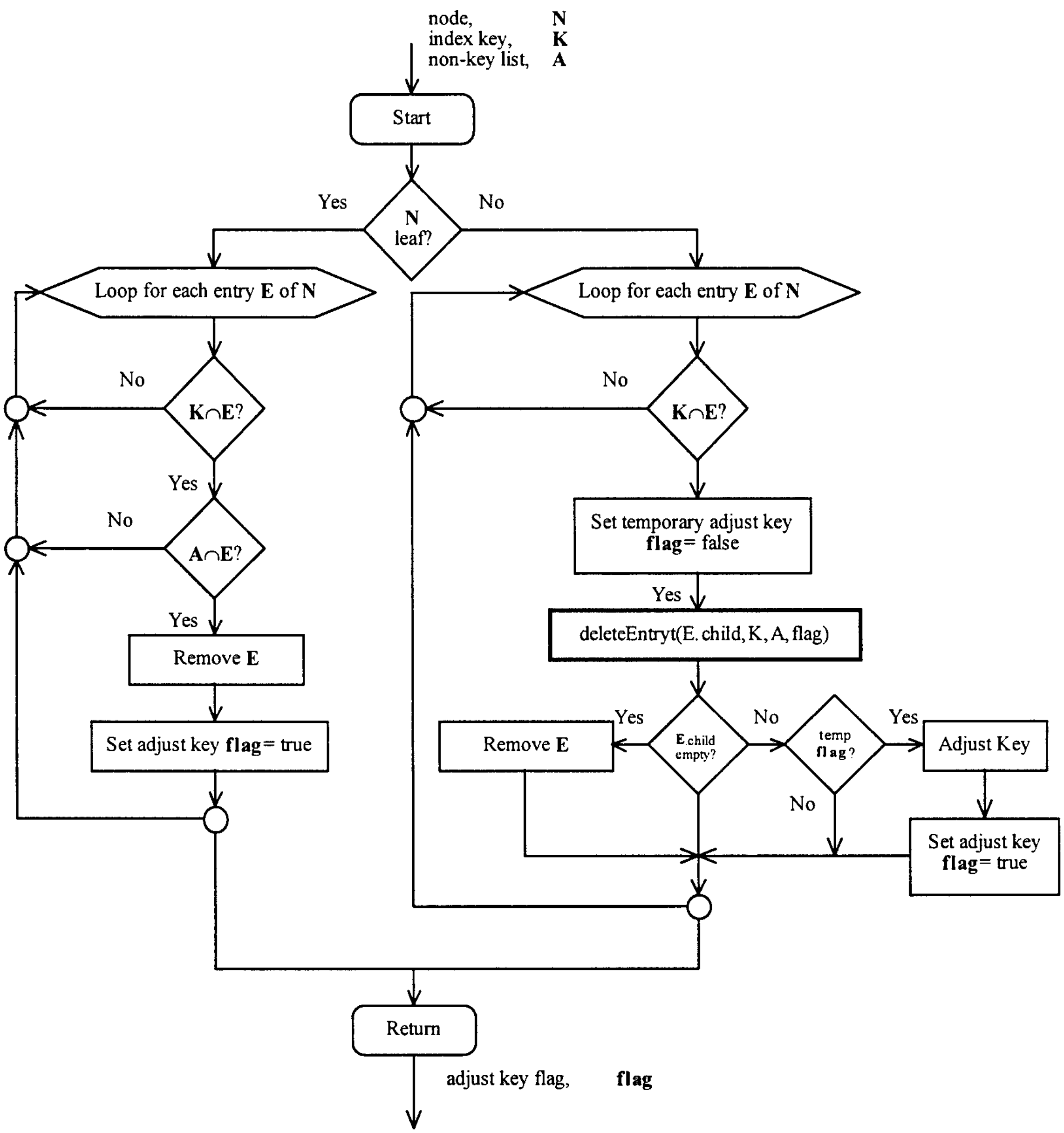


Figure A-27: Delete Entry Control Flow Diagram

The *deleteEntry* routine will correctly implement the process for all nodes except the root node. The situation is similar to the *insert* routine described earlier and a similar solution is chosen for the implementation. A special routine called *delete*, shown in Figure A-28, will implement the command when the current node is the root node. Thus, the *main* routine should only delete data objects through the *delete* routine and not the *deleteEntry* routine. The first part of the *delete* routine simply calls the *deleteEntry* routine above. Depending on whether the node is a leaf or non-leaf node, each entry on the node is checked for consistency and processed, recursively calling *deleteEntry* as required, thus the entire index is processed. Upon return from the recursive *deleteEntry*, the index is in one of three states as described above for any node, but, since the root node has no parent, the information is not processed in any way. However, if the input search key and non-key values caused all of the data objects referenced by the index to be removed, then the root node will have an empty entry list and may have a specified level greater than the leaf level. This case is easily handled by simply setting the root node level to the leaf level and continuing. If the root node has only one child and is not a leaf node, the child becomes the new root as required by the R-Tree specification. The single child check is repeated until the condition is satisfied.

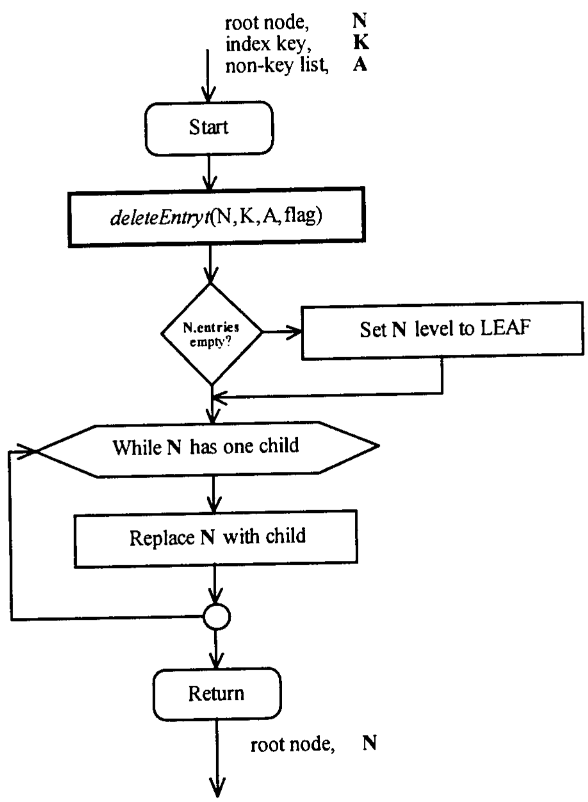


Figure A-28: Delete Control Flow Diagram

A.5 INPUT & OUTPUT

The Input & Output module localizes the system level I/O routines and allows separate development of the database algorithms without requiring a complete application "wrapper". The first section reviews the input specification requirements which includes a discussion of all four commands. The second section reviews the output specification requirements. The final section discusses the implementation of the module.

A.5.1 Input Specification shortcut keys

The input for each test of this benchmark consists of one data set. All of the data sets share a common format. Each set is a 8-bit ASCII character file and consists of a series of sequentially issued commands delimited by a carriage return, i.e., each line of the file represents a separate command. Table A-2 gives the command operations, the character code used to designate the command, the data placed after the command code on the rest of the line, the return expected from the application, and a brief description of the operation.

Table A-2: Command Operations

Command	Code	Line Elements	Return	Description
Initialization	0	Fan Size	NULL	Initializes the index by specifying the fan of the tree.
Insert	1	Object Type Key Attribute Key Attribute : Key Attribute Non-Key Attribute Non-Key Attribute : Non-Key Attribute	NULL	Insert new entry into database. See below for discussion of the Object Type and key and non-key attributes.
Query	2	Attribute Code Attribute Value Attribute Code Attribute Value : Attribute Code Attribute Value	Data Object List	Return all data objects which are consistent with the input attributes specified. Note that attribute codes and values always appear as pairs.
Delete	3	Attribute Code Attribute Value Attribute Code Attribute Value : Attribute Code Attribute Value	NULL	Delete all data objects which are consistent with the input attributes specified. Note that attribute codes and values always appear as pairs.

Each data object has a set of attributes, where the first eight attributes are used by the R-Tree index as the key and represent two points which specify a hyper-cube. Each point consists of four 32-bit IEEE-formatted floating-point numbers representing a four-dimensional Euclidean point denoted as the T-position, X-position, Y-position, and Z-position. Thus, the index key, which consists of a "lower" and "upper" point, is eight 32-bit floating-point numbers.

The total number of attributes assigned to a data object is the sum of the key and non-key attributes, where the number of non-key attributes for a given data object is determined by the Object Type and is given in Table A.1. The Object Type used by the *Insert* command specifies which of the three types of objects (Small, Medium, and Large) is being inserted by the operation. Table A.1 gives the character/byte code and the number of non-key attributes for each data object type

Data objects differ by the number of non-key attributes assigned to each. Each non-key data attribute has an identical format which primarily consists of an 8-bit NULL-terminated ASCII character sequence of maximum length 1024. Table A.1 gives the number of non-key attributes assigned to each object type, and the database should be able to handle all three types of data object, in any permutation. Note that the object type specification is placed at the beginning of the *Insert* command as a convenience, since the number of attributes can be determined by reading until the next carriage return, i.e., the end of the command.

The *Delete* and *Query* commands each reference a specified attribute by means of an Attribute Code. Table A.3 gives the attribute code sequence for both the key and non-key attributes. Also, each attribute is assigned a type and, if applicable, a name and units.

Table A.3: Attribute Codes and Descriptions

		Attribute Code	Name	Type
Key Attributes	Lower Point	0	T	*float*
		1	X	*float*
		2	Y	*float*
		3	Z	*float*
	Upper Point	4	T	*float*
		5	X	*float*
		6	Y	*float*
		7	Z	*float*
		8	property	*string*
		9	property	*string*

Non-Key Attributes		Small	:	:	:
			16	property	*string*
			17	property	*string*
		Medium	18	property	*string*
			19	property	*string*
			:	:	:
			23	property	*string*
			24	property	*string*
	Large		25	property	*string*
			26	property	*string*
			:	:	:
			49	property	*string*
			50	property	*string*

The formal definition of each command input line is described in the specification and is not repeated here except to note the idea of "wild card" values for missing key and non-key attributes for both the Query and Delete commands. A wild card value will match any other value when compared for a consistency check, i.e., a wild card value for the lower T hyper-point will match any other value for the lower T hyper-point. The wild card values are implemented in the two routines *getQueryCommand* and *getDeleteCommand* which are described in following sections.

A.5.2 Output Specification

The only command which produces a response from the index is the *Query* command. The output of the database are the responses to each *Query* operation.

The response to a *Query* operation consists of a set of data objects which are consistent with the *Query*. Each data object in a response is represented as the list of its attributes in order defined by Table A.3. The list of attributes is written to a 8-bit ASCII character file where each attribute is space delimited and with each list carriage return delimited. The format is very similar to the *Insert* operation format for the input data sets with the only difference being the omission of the command code. The set of data objects returned by a *Query* must be placed in the output file continuously, i.e., in adjacent lines, although the order of the set is not constrained.

A.5.3 Implementation

A complete function hierarchy for the Input & Output module is given in Figure A-29. The two routines, *openFiles* and *closeFiles*, are meant to partially initialize and exit the module respectively. Three files should be opened/closed by these routines: the input dataset file, the output response file, and the metric file. The names of each file will be passed as command-line parameters, with the default names being the standard input for the dataset file, standard output for the output file, and standard error for the metric file. The input and output files contents and formats are discussed in the previous sections. The metric file is also used to display/store the metric information for an individual execution of the baseline source code. Input to the database application is done by five routines. Four of the routines correspond directly with the command type required by the DIS Benchmark Specification: Init, Insert, Query, and Delete. The fifth routine reads the command code which specifies which command is being read.

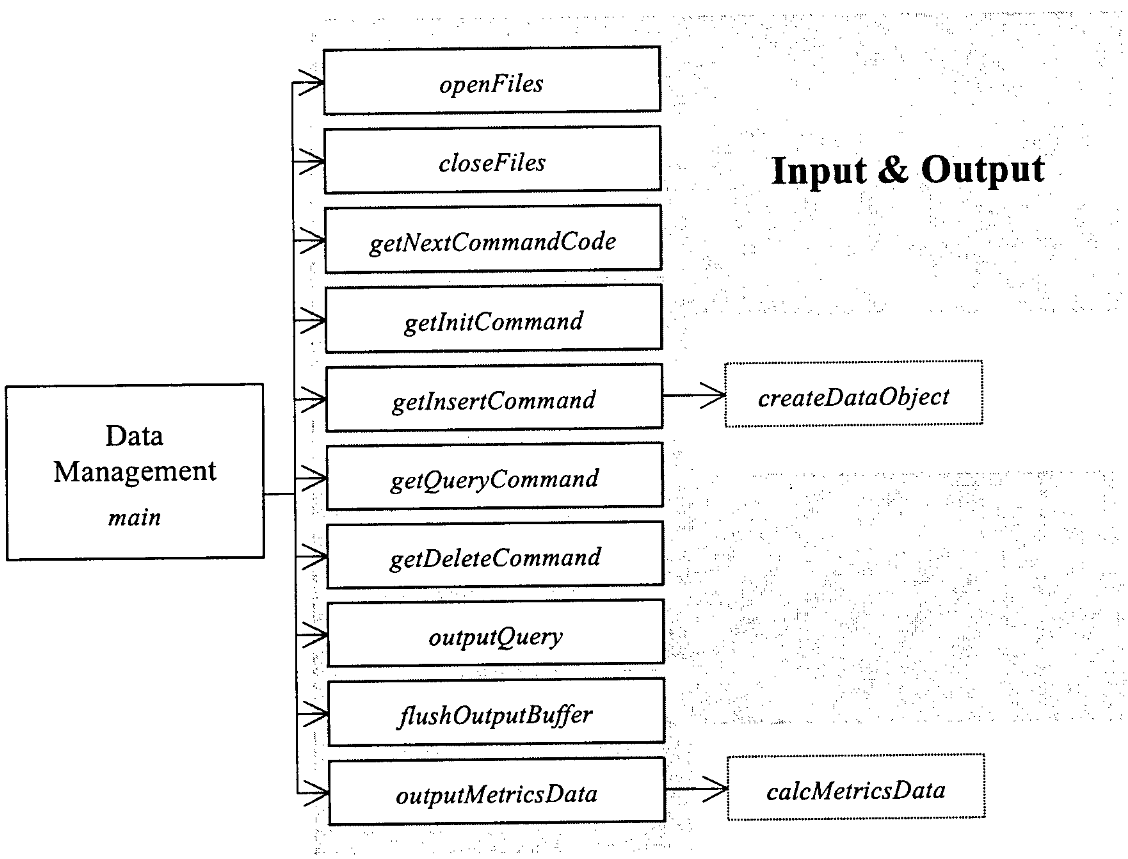


Figure A-29: Input & Output Module Function Hierarchy

Each command has a specified command code (see Table A-2) which is always the first parameter of the command. The action taken by the application will branch depending on which command was read so a routine to read this code is implemented and is called *getNextCommandCode*. The routine will only read the command code and will leave the input stream pointer at a location where subsequent routines can read the rest of the command. If an invalid code is read, the routine will attempt to "drop down" to the next command until a valid code is reached. Also, the routine should return a specified condition to indicate that the end of the input file has been reached which means the data set has been completely processed.

The Init command is read by the *getInitCommand* routine. As mentioned previously, the initialization command is different from the rest of the database commands since it appears first and only once for each data set as well as its unique format. The *getInitCommand* is called by *main* as part of the application initialization.

The Insert command is read by the *getInsertCommand* routine which reads the entire command from the input dataset and returns a data object which can be directly inserted into the index. This is accomplished by allowing the routine to access the Database module routine *createDataObject*. This means that any change to the database data object structure may cause changes in the *getInsertCommand* routine. This violates the idea of a module, but is allowed to reduce the size of the baseline source code and complexity of the resulting benchmark application.

The Query command is read by the *getQueryCommand* routine which reads the entire command from the input and returns two search values ready for the *query* routine: an index key and a non-key attribute list. Missing values in the command for both key and non-key values are defaulted to wild-cards, i.e., will match with anything. Default wild-card values will be defined and inserted into both search values inside the *getQueryCommand* routine. This could be done outside of the routine at the main level or inside the *query* routine, but the "missing" value specification is judged to be more of an input format rather than a database behavior for the benchmark.

The Delete command is read by the *getDeleteCommand* routine which reads the entire command from the input and returns two search values ready for the *delete* routine: an index key and a non-key attribute list. The use of wild-card values and reasoning is identical for the *getDeleteCommand* as for the *getQueryCommand*.

The final three routines of the Input & Output module handle the output of the application. The first, *outputQuery*, places the query response into the output buffer, which does not necessarily write the output to the screen or a file. Instead, another routine, *flushOutputBuffer*, is used to write the buffer to the screen/file. This is done to conform to the benchmark specification when timing the Query command (see section A.6). The buffer will be limited in size, so a condition may arise where the buffer would need to be flushed before its usual placement in the execution loop (see Figure A-3). Although this will degrade the time of the query which caused this premature flush, the condition is expected to be rare enough that the overall metric statistics will be only slightly affected. The final routine, *outputMetricsData*, displays the metrics data. The routine first calls the metrics module routine *calcMetricsData* which calculates the separate metric data, averages, deviations, etc. This means that any change to the metric structures may cause changes in the *outputMetricsData* routine.

A.6 METRICS

The primary metric associated with the Data Management benchmark is total time for accurate completion of a given input data set. A series of secondary metrics are the individual times of the command operations: *Insert*, *Delete*, and *Query*. Best, worst, average, and standard deviation times are to be reported for all operations.

The time for the *Insert* and *Delete* commands to complete is defined as the time difference between the time immediately before the command is placed in the database input queue and the time immediately before the next command is placed in the same input queue. This time difference is essentially the rate at which each line of the input data set is read. The time for a *Query* command to complete is defined as the time difference between the time immediately before the command is placed in the input queue to the time immediately after the response is placed in the output queue.

The Metrics module is intended to separate the timing issues from the rest of the application, i.e., different system timing routines should only have to change the routines in the Metrics module without effecting the Database module. The tasks required by the Metrics module fall into four different categories. The first is an initialization step which should determine the starting time of the application and set any needed flags or values for later timing calculations. The second category is the "setting" of flags for repeated passes through loops and calculations, i.e., some form of time marker would need to be set each time a command is processed. The third category would be to update the values of the metric statistical data using the flags and time markers set by the second category. The final category would be the calculation of the specific metric values required by the benchmark. The reason that the third and fourth categories are separated is for efficiency, since the third category of updating values could also calculate the required metrics. However, the update category would need to be executed every time the second or "setting" category was changed in order to keep accurate metric values. The extra calculations would be "wasted" until the entire data set was completed.

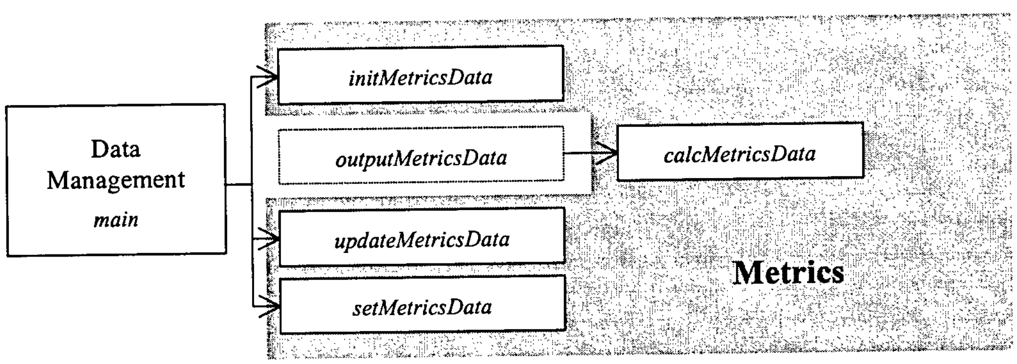


Figure A-30: Metrics Module Function Hierarchy

The Metrics module implements four routines which correspond directly with the four categories described above: *initMetricsData*, *setMetricsData*, *updateMetricsData*, and *calcMetricsData*.

The implementation of the Metrics module involves a system request for the current time. The ANSI/ISO C standard library provides a routine, *time*, which returns the time with a resolution of seconds. This fidelity may be accurate for the primary metric of total time required to process the data set, but will probably not be sufficient for the individual command metrics. Another routine, *gettimeofday*, is part of the BSD UNIX standard and returns the time with a resolution of microseconds. This fidelity would yield the necessary accuracy, but storage of intermediate times may become difficult since the *gettimeofday* routine returns a structure of two longs and the metric statistical calculations will require time value arithmetic, i.e., subtraction, multiplication, square root, etc. Also, the *gettimeofday* routine is not part of the ANSI/ISO C standard. A compromise is made for the baseline application which consists of wrapping the system timing routines in a timer routine called *getTime* which returns the current system time in milliseconds since the program began execution. A preprocessor variable will be used to allow the *getTime* routine to use either the ANSI/ISO C standard library routine *time*, or the BSD UNIX *gettimeofday* routine. The *getTime* routine returns milliseconds from program start, to allow the truncated *gettimeofday* return values to be stored in a single long integer. This does mean a loss of fidelity when using the *gettimeofday* routine, but milliseconds should provide an accurate enough measure for the metric calculations and the ease of implementation, a single long integer rather than a structure of two long integers, outweighs the reduced fidelity. The values returned by the *time* routine are "promoted" to milliseconds in the *getTime* routine without increasing the accuracy of the measurements. Any changes to the timing routine and system level time calls, should only affect the *getTime* function which should allow users to quickly and safely modify the baseline source to their own systems.

A.7 REFERENCES

[AAEC-1] "Data-Intensive Systems Benchmark Suite," MDA972-97-C 0025, Atlantic Aerospace Electronics Corp., Waltham, MA 02451, Spring 1999.

[AAEC-2] "DIS Benchmark C Style Guide: C Style and Coding Standards for the DIS Benchmark Suite," MDA972-97-C 0025, Atlantic Aerospace Electronics Corp., Waltham, MA 02451, December 28, 1998.

[Guttman] Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOID*, pp. 47-57, June 1984.

A.8 PSEUDO-CODE

This part of the document lists the pseudo-code for the baseline application. The first section contains the main routine, a listing of the command structures and variables, and support routines used by the main and modules. The lower routines are listed under the appropriate module sections and include a complete routine description, comments, pseudo-code, and brief description of a unit test if applicable.

A.9 COMMON TYPES AND STRUCTURES

```
typedef long int        Int;
typedef float           Float;
typedef char            Char;
typedef unsigned char   Uchar;

typedef long int        Time;
typedef struct timeval  Time;
/*
 * Command type which includes a NONE for no commands left
 */
typedef enum
{
            INIT    = 0,
            INSERT  = 1,
            QUERY   = 2,
```

```
            DELETE  = 3,
            NONE    = 4,
            INVALID = 5
} CommandType;

/*
 * Data Object Types for determining number of non-key attributes
 */
typedef enum
{
            SMALL  = 1,
            MEDIUM = 2,
            LARGE  = 3
} DataObjectType;

/*
 * Data Object Attribute which uses union to handle both
 * key and non-key values
 */
typedef struct
{
            union {
                  Float         key;
                  Char          *nonKey;
            }                   value;
} DataObjectAttribute;

/*
 * Data Attribute.  A subclass of DataObjectAttribute which adds
 * an attribute code to the object attribute structure.  The
 * structure is used by the query and delete commands for key and
 * non-key searches.  Any code value between MIN_ATTRIBUTE_CODE
 * and NUM_OF_KEY_ATTRIBUTES represent a key value, i.e., Float.
 * Any code value between NUM_OF_KEY_ATTRIBUTES + 1 and
 * MAX_ATTRIBUTE_CODE represents a non-key value, i.e., Char *.
 */
typedef struct
{
            Int                       code;       /* code for attribute
            */
            DataObjectAttribute       attribute;  /* attribute for code
            */
} DataAttribute;

/*
 * Data Object
 *     - basic storage unit of database which is referenced by R-
 *       Tree index.
 *     - contains key and non-key attributes in an array
 *     - number of attributes is determined by DataObjectType
 *       specifier
 */
typedef struct
{
            DataObjectType            type;       /* type of object  */
            DataObjectAttribute       *attribute; /* attribute array */
} DataObject;
```

```
/*
 * Index Node
 */
typedef struct
{
            Int         level;        /* level of tree where */
                                      /* node resides        */
            List        entries;      /* list of data entries */
                                      /* for this node        */
} IndexNode;

/*
 * Index point structure
 */
typedef struct
{
            Float       T;
            Float       X;
            Float       Y;
            Float       Z;
} IndexPoint;

/*
 * Index key structure - defines a hyper-cube from two points
 */
typedef struct
{
            IndexPoint  lower;        /* lower point of hyper-cube */
            IndexPoint  upper;        /* upper point of hyper-cube */
} IndexKey;

/*
 * Index Entry
 */

typedef struct
{
            /* pointer to child which is either                */
            /*    1) Another node in index                     */
            /*    2) Data object which is stored               */
            union {
                  IndexNode   *node;
                  DataObject  *dataObject;
            }           child;
            IndexKey    key;  /* index key for the entry */
} IndexEntry;

/*
 * Command metric structure for timing statistics
 */
typedef struct
{
            /*
             * The four following members are for internal use and
             * should never be accessed outside of the metrics
            routines.
```

```
     */
    Time        lastTimeMark;      /* time mark for command */
                                   /* time duration calc    */
    Int         numOfCommands;     /* number of commands */
                                   /* (for avg)          */
    Time        sum;               /* sum of the individual */
                                   /* command times         */
    Time        sumSquares;        /* sum of the squares of */
                                   /* the individual times */
    /*
     * The four following members are for external use and
     * represent the metric results for this command.
     */
    Time        worst;             /* worst time recorded */
                                   /* for this command */
    Time        best;              /* best time recorded for */
                                   /* this command */
    Time        avg;               /* average time for this */
                                   /* command */
    Time        deviation;         /* standard deviation for */
                                   /* this command */

} CommandMetric;

/*
 * Metric structure
 */
typedef struct
{
        /*
         * Metric Times
         */
        Time totalTime;    /* total time to complete data set  */
        Time inputTime;    /* total time for input of data set */
        Time outputTime;   /* total time for output of queries */
        /*
         * Individual Command Metrics
         */
        CommandMetric insertCommandMetric;  /* insert metric  */
        CommandMetric queryCommandMetric;   /* query metric   */
        CommandMetric deleteCommandMetric;  /* delete metric  */
        /*
         * The time for a command to complete is a complete "loop"
         * through the command process, i.e., reading, application
         * to index, output if any, etc.  To do this properly, a
         * flag or value must be set each time through the loop,
        via
         * the setMetricsData routine which will be used by the
         * updateMetricsData routine to update the proper
         * CommandMetric structure.
         */
        CommandType lastCommand /* command used for             */
                                /* determining which command    */
                                /* metric to update             */

 } Metric;

 /*
  * Non-Key Attribute Parameters
```

```
 */
#define MAX_SIZE_OF_ATTRIBUTE       1024;
#define NUM_OF_SMALL_ATTRIBUTE        18;
#define NUM_OF_MEDIUM_ATTRIBUTE       25;
#define NUM_OF_LARGE_ATTRIBUTE        51;
#define NUM_OF_KEY_ATTRIBUTES          8;
#define MIN_ATTRIBUTE_CODE             0;
      #define MAX_ATTRIBUTE_CODE             50;

enum {
            LOWER_POINT_T = 0,
            LOWER_POINT_X = 1,
            LOWER_POINT_Y = 2,
            LOWER_POINT_Z = 3,
            UPPER_POINT_T = 4,
            UPPER_POINT_X = 5,
            UPPER_POINT_Y = 6,
            UPPER_POINT_Z = 7
};

/*
 * Wild-Card Values
 * ( min's and max's determined by IEEE 754 Spec )
 */
#define MINIMUM_VALUE_OF_FLOAT      -3.40282347e38;
#define MAXIMUM_VALUE_OF_FLOAT       3.40282347e38;
#define MINIMUM_VALUE_OF_INT        -2145483647;
#define MAXIMUM_VALUE_OF_INT         2145483647;

/*
 * Index (R-Tree) Parameters
 */
#define MINIMUM_FAN_SIZE            2
```

A.10 MAIN

Name:	main
Input:	input file name (option w/default being stdin)
Output:	database response output file (option w/default being stdout) metrics output file (option w/default being stderr)
Description:	Main routine which handles initialization of modules, and interface between the I/O, Database, and Metrics.
Calls:	openFiles() closeFiles() getInitCommand() getInsertCommand() getQueryCommand() getDeleteCommand() getNextCommandCode() outputQuery() outputMetricsData() createIndexNode()

deleteIndexNode()
insert()
query()
delete()
initMetricsData()
updateMetricsData()
setMetricsData()
errorMessage()
System:

```
Get command-line integer, argc          /* # of line arguments      */
Get command-line arguments, argv        /* command-line arguments   */
Define Metric struct, metrics           /* timing metric            */
Define FILE pointer, *inputFile         /* input file pointer       */
Define FILE pointer, *outputFile        /* output file pointer      */
Define FILE pointer, *metricFile        /* metric file pointer      */
Define index node pointer, *root        /* root node pointer        */
Define Time, lastTimeMark               /* timing variable          */
Define integer, code                    /* database command         */
Define integer, fan                     /* fan for index            */
Define integer, error                   /* error flag for routines */
/*
 * Set defaults for input, output, and metric files
 */
Set inputFile  = stdin
Set outputFile = stdout
Set metricFile = stderr

/*
 * Parse command-line arguments
 */
If argc > 1 Then
            /*
             * dis_datamanagement -v
             *     -i inputFileName
             *     -o outputFileName
             *     -m metricFileName
             */
EndIf

/*
 *
 *
 *         Begin Initialization of Database Application
 *
 *
 */

/*
 * Initialize Metrics
 */
Call initMetricsData( metrics )

/*
 * Initialize I/O
```

```
 */
Call error = openFiles( inputFile, outputFile, metricFile )
If error != OPEN_FILES_SUCCESS Then
            /*
             * FATAL: can't proceed if unable to open files
             */
            Return ERROR
EndIf

/*
 * Initialize Database by creating the first root node and getting
 * the specified fan from the input file.
 */
Call root = createIndexNode( LEAF )
If root == NULL Then
            /*
             * FATAL: can't proceed if unable to create root node
             */
            Return ERROR
EndIf

/*
 * Get command code of first command in file.  This command MUST
 * be an INIT command as specified by the DIS Benchmark
 * Specification.  Any command other than INIT will result in a
 * termination of the application.
 */
Call error = getNextCommandCode( inputFile, code )
If error == GET_NEXT_COMMAND_CODE_SUCCESS && code == INIT Then
            /*
             * Read INIT command to get the fan of the index
             */
            Call error = getInitCommand( inputFile, fan )
            If error == GET_INIT_IO_ERROR ||
               error == GET_INIT_EARLY_EOF_OR_EOL_ERROR ||
               error == GET_INIT_INVALID_FAN Then
                  /*
                   * FATAL: can't proceed without a valid init command
                   */
                  Return ERROR
            EndIf
Else If error == GET_NEXT_COMMAND_CODE_IO_ERROR Then
            /*
             * FATAL: can't proceed if unable to read code for
             * initialization command
             */
            Return ERROR
Else If code != INIT Then
            /*
             * FATAL: can't proceed if first command is not INIT
             */
            Return ERROR
EndIf

/*
 *
 *
```

```
 *        Begin Main Body of Database Application
 *
 *
 */

/*
 * Process commands in inputFile until done
 */
Call error = getNextCommandCode( inputFile, code )
If error == GET_NEXT_COMMAND_CODE_SUCCESS Then
            /*
             * Empty
             */
Else If error == GET_NEXT_COMMAND_IO_ERROR Then
            /*
             * FATAL: low-level I/O error reading next command code
             */
            Return ERROR
EndIf

Loop while code != NONE Then
            /*
             * Insert
             */
            If code == INSERT Then
                  /*
                   * Read insert command
                   */
                  Define data object, dataObject
                  Call error = getInsertCommand( inputFile, dataObject
                  )
                  If error == GET_INSERT_SUCCESS Then
                        /*
                         * Pass entry to index for insertion
                         */
                        Call error = insert( root, dataObject, fan )
                        If error == INSERT_SUCCESS Then
                              /*
                               * Empty
                               */
                        Else If error == INSERT_INSERT_ENTRY_FATAL Then
                              /*
                               * FATAL: the index has changed and an
                               * error occurred during the inserting of
                               * an entry.
                               */
                              Return ERROR
                        Else If error == INSERT_INSERT_ENTRY_NON_FATAL
                        Then
                              /*
                               * NON FATAL: the index has not changed
                               * and an error has occurred. The current
                               * implementation notifies the user of
                              the
                               * insert failure and attempts to
                              proceed.
                               */
```

```
        Else If error == INSERT_ALLOCATION_FAILURE Then
              /*
               * FATAL: the index has changed and there
               * was no more memory to grow the tree.
               */
              Return ERROR
        EndIf
    Else If error == GET_INSERT_IO_ERROR Then
          /*
           * FATAL: low-level I/O error occurred during
           * read. No recovery.
           */
          Return ERROR
    Else If error == GET_INSERT_EARLY_EOF_OR_EOL_ERROR
       Then
          /*
           *
           */

    Else If error == GET_INSERT_UNKNOWN_DATA_OBJECT_TYPE
       Then
          /*
           *
           */

    Else If error == GET_INSERT_ALLOCATION_ERROR Then
          /*
           *
           */

    EndIf
/*
 * Query
 */
Else If code == QUERY Then
    Define index entry list,      solutionSet
    Define index key,             searchKey
    Define non-key list,          searchNonKey

    /*
     * Get Query command from input
     */
    Call error = getQueryCommand( inputFile, searchKey,
       searchNonKey )
    If error == GET_QUERY_SUCCESS Then
          /*
           * Pass searchKey and searchNonKey to index to
           * find solutionSet via a query
           */
          Call error = query( root, searchKey,
             searchNonKey, solutionSet )
          If error == QUERY_SUCCESS Then
                /*
                 * Query command gives an output solution
                 * set which is placed in the output
                 * buffer for later flushing.
                 */
```

```
                Call outputQuery( solutionSet )
            Else If error == QUERY_INVALID_KEY_SEARCH_VALUE
               Then
                /*
                 * The key value used for search is
                 * invalid, i.e., bounds on lower-upper
                 * cube are reversed, etc.  Current
                 * implementation warns user and
                 * continues.
                 */
            Else If error ==
               QUERY_INVALID_NON_KEY_ATTRIBUTE_CODE Then
                /*
                 * One of the non-key attribute values
                 * (character string) is invalid, i.e.,
                is
                 * empty(NULL).  Current implementation
                 * warns user and continues.
                 */
            EndIf
        Else If error == GET_QUERY_IO_ERROR Then
            /*
             * FATAL: low-level I/O error occurred during
             * read. No recovery.
             */
            Return ERROR
        Else If error ==GET_QUERY_EARLY_EOF_OR_EOL_ERROR Then
            /*
             *
             */

        Else If error == GET_QUERY_INVALID_CODE_ERROR Then
            /*
             *
             */

        Else If error == GET_QUERY_ALLOCATION_ERROR Then
            /*
             *
             */

        EndIf
    /*
     * Delete
     */
    Else If code == DELETE Then
        Define index key,              searchKey
        Define non-key list,           searchNonKey

        /*
         * Get Delete command from input
         */
        Call error = getDeleteCommand( inputFile, searchKey,
           searchNonKey )
        If error == GET_DELETE_SUCCESS Then
            /*
             * Pass searchKey and searchNonKey to index to
```

```
             * delete data objects which intersect with the
             * search values.
             */
            Call error = delete( root, searchKey,
               searchNonKey, fan )
            If error == DELETE_SUCCESS Then
                  /*
                   * Delete command does not give output.
                   */
            Else If error ==
               DELETE_INVALID_KEY_SEARCH_VALUE Then
                  /*
                   * The key value used for search is
                   * invalid, i.e., bounds on lower-upper
                   * cube are reversed, etc.  Current
                   * implementation warns user and
                   * continues.
                   */
            Else If error ==
               DELETE_INVALID_NON_KEY_ATTRIBUTE_CODE Then
                  /*
                   * One of the non-key attribute values
                   * (character string) is invalid, i.e.,
                  is
                   * empty(NULL).  Current implementation
                   * warns user and continues.
                   */
            EndIf
      Else If error == GET_DELETE_IO_ERROR Then
            /*
             * FATAL: low-level I/O error occurred during
             * read. No recovery.
             */
            Return ERROR
      Else If error ==GET_DELETE_EARLY_EOF_OR_EOL_ERROR
      Then
            /*
             *
             */

      Else If error == GET_DELETE_INVALID_CODE_ERROR Then
            /*
             *
             */

      Else If error == GET_DELETE_ALLOCATION_ERROR Then
            /*
             *
             */

      EndIf
/*
 * Unknown or NONE command code read
 */
Else
      error handler( WARNING: Unknown or invalid command
         type read )
```

```
            EndIf

            /*
             * Get the next command code for next command processing
             */
            Call error = getNextCommand( inputFile, command )
            If error == GET_NEXT_COMMAND_CODE_SUCCESS Then
                  /*
                   * Next command ready-to-go
                   */
            Else If error == GET_NEXT_COMMAND_CODE_IO_ERROR Then
                  /*
                   * FATAL: low-level I/O error reading next command
                   */
                  Return ERROR
            EndIf
EndLoop

/*
 * Exit Database
 */
Call deleteIndexNode( root )

/*
 * Exit Metrics - via the outputMetricsData routine
 */
Call error = outputMetricsData( metricFile, metrics )
If error == OUTPUT_METRIC_SUCCESS Then
            /*
             * Empty
             */
Else If error == OUTPUT_METRIC_SUCCESS Then
            /*
             * Failure during writing of metrics.  At the end of
             * application, so do nothing.
             */
EndIf

/*
 * Exit I/O
 */
Call closeFiles( inputFile, outputFile, metricFile )

/*
 * Done
 */
Return SUCCESS
```

A.11 DATABASE

A.11.1 <u>chooseEntry</u>

Name:	chooseEntry
Input:	node with list of entries to choose from, node

	entry trying to choose with, entry
Output:	entry within node which minimizes penalty, minPenaltyEntry
Return:	index entry, NULL if node is empty
Description:	Determines which entry of input node to add new entry. Chosen entry provides minimum penalty which is the increase of the total hyper-cube volume. The only error possible is if the node provided to choose from is empty.
Calls:	errorMessage() penalty() System:

```
Get index node, node          /* node to choose from  */
Get index entry ptr, entry    /* entry to choose with */
/*
 * Find entry of node which has the lowest penalty
 * for adding entry
 */
If node.entries == EMPTY Then
            Call errorMessage( WARNING: empty node )
            Set minPenaltyEntry = NULL
Else
            /*
             * Find entry of node which has the lowest penalty for
             * adding entry
             */
            Define index entry minPenaltyEntry
            Define Float minPenalty
            Set minPenaltyEntry to first entry of node.entries
            Set minPenalty = penalty( minPenaltyEntry, entry )
            Loop for each entry tempEntry of node excluding first entry
                  Define float tempPenalty
                  Set tempPenalty = penalty( tempEntry, entry )
                  If tempPenalty < minPenalty Then
                        Set minPenaltyEntry = tempEntry
                        Set minPenalty      = tempPenalty
                  EndIf
            EndLoop
EndIf
/*
 * Done
 */
Return minPenaltyEntry
```

Test:

A.11.2 consistentKey

Name:	consistentKey
Input:	index key, **A** index key, **B**
Output:	flag indicating intersection
Return:	integer, TRUE or FALSE
Description:	Returns Boolean value indicating if the two input index keys intersect.

Calls:

System:

```
/*
 * Consistent check for key attributes
 */
Int                         /* return value either TRUE or FALSE */
consistentKey(
      IndexKey    A,        /* first index key                   */
      IndexKey    B         /* second index key                  */
)
{
            /*
             * Check T bounds
             */
            if ( A.lower.T > B.upper.T ||
                 B.lower.T > A.upper.T )
            {
                  return ( FALSE );
            }

            /*
            * Check X bounds
             */
            if ( A.lower.X > B.upper.X ||
                 B.lower.X > A.upper.X )
            {
                  return ( FALSE );
            }

            /*
            * Check Y bounds
             */
            if ( A.lower.Y > B.upper.Y ||
                 B.lower.Y > A.upper.Y )
            {
                  return ( FALSE );
            }

            /*
            * Check Z bounds
             */
            if ( A.lower.Z > B.upper.Z ||
                 B.lower.Z > A.upper.Z )
            {
                  return ( FALSE );
            }

            /*
             * Passed all bounds test
             */
            return ( TRUE );
}
```

Test:

A.11.3 consistentNonKey

Name:	consistentNonKey
Input:	character string, **A**
	character string, **B**
Output:	integer, TRUE or FALSE
Return:	Boolean value flag
Description:	Searches for string **B** in string **A** returning TRUE if **A** contains **B** and FALSE otherwise.
Calls:	
	System: strchr()

```
/*
 * Consistent check for non-key attributes
 */
Int                 /* return value either TRUE or FALSE        */
consistentNonKey(
      Char  *A,     /* first non-key value                      */
      Char  *B      /* second non-key value                     */
)
{
            /*
             * Use standard library function for sub-string comparison
             */
            if ( ::strchr( A, B ) == NULL )
            {
                  return ( FALSE );
            }

            /*
             * Passed test
             */
            return ( TRUE );
}
```

Test:	

A.11.4 createDataObject

Name:	createDataObject
Input:	DataObjectType, **type**
Output:	data object created, **dataObject**
Return:	DataObject pointer, or
	NULL (if allocation failed)
Description:	Routine creates data object of specified type by allocating appropriate amount of memory for number of non-key attributes. Note that memory is not allocated for the individual non-key character strings since their specific size is not known before hand. The values for the data object are also filled with default values which produces an index key of maximum possible volume and sets each non-key character string to NULL.

Calls:	errorMessage() System: malloc()

```
Get DataObjectType, type /* type of data object to create */
Define DataObject pointer, dataObject /* data object to create */
/*
 * Create data object
 */
Set dataObject = new DataObject
If dataObject == NULL Then
            Call errorMessage( can't create new data object )
Else
            /*
             * Set default index key values
             */
            Set dataObject.key.lower.T = MINIMUM_VALUE_OF_FLOAT
            Set dataObject.key.lower.X = MINIMUM_VALUE_OF_FLOAT
            Set dataObject.key.lower.Y = MINIMUM_VALUE_OF_FLOAT
            Set dataObject.key.lower.Z = MINIMUM_VALUE_OF_FLOAT
            Set dataObject.key.upper.T = MAXIMUM_VALUE_OF_FLOAT
            Set dataObject.key.upper.X = MAXIMUM_VALUE_OF_FLOAT
            Set dataObject.key.upper.Y = MAXIMUM_VALUE_OF_FLOAT
            Set dataObject.key.upper.Z = MAXIMUM_VALUE_OF_FLOAT

            /*
             * Allocate non-key attribute array
             */
            If type == SMALL Then
                  Set dataObject.type = SMALL

                  Set dataObject.attributes = new Char * [
                     NUM_OF_SMALL_ATTRIBUTE ]
                  /*
                   * Check
                   */
                  If dataObject.attributes == NULL Then
                        Call errorMessage( can't create non-key
                           attribute array for new data object )
                        Delete dataObject
                        Set dataObject = NULL
                  Else
                        /*
                         * Set default non-key values
                         */
                        Loop for i = 0 to NUM_OF_SMALL_ATTRIBUTE - 1
                              Set dataObject.attributes[ i ] = NULL
                        EndLoop
                  EndIf
            Else type == MEDIUM Then
                  Set dataObject.type = MEDIUM

                  Set dataObject.attributes = new Char * [
                     NUM_OF_MEDIUM_ATTRIBUTE ]
                  /*
                   * Check
                   */
                  If dataObject.attributes == NULL Then
```

```
                Call errorMessage( can't create non-key
                   attribute array for new data object )
                Delete dataObject
                Set dataObject = NULL
          Else
                /*
                 * Set default non-key values
                 */
                Loop for i = 0 to NUM_OF_MEDIUM_ATTRIBUTE - 1
                      Set dataObject.attributes[ i ] = NULL
                EndLoop
          EndIf
    Else If type == LARGE Then
          Set dataObject.type = LARGE

          Set dataObject.attributes = new Char * [
             NUM_OF_LARGE_ATTRIBUTE ]
          /*
           * Check
           */
          If dataObject.attributes == NULL Then
                Call errorMessage( can't create non-key
                   attribute array for new data object )
                Delete dataObject
                Set dataObject = NULL
          Else
                /*
                 * Set default non-key values
                 */
                Loop for i = 0 to NUM_OF_LARGE_ATTRIBUTE - 1
                      Set dataObject.attributes[ i ] = NULL
                EndLoop
          EndIf
    Else
          Call errorMessage( invalid object type to create )
          Delete dataObject
          Set dataObject = NULL
    EndIf
EndIf

/*
 * Done
 */
Return dataObject
```

Test:

A.11.5 createIndexEntry

Name:	createIndexEntry
Input:	none
Output:	new index entry, **entry**
Return:	IndexEntry pointer, or NULL (if allocation failed)

Description:	Routine creates a new index entry using system allocation routine. Returns pointer to new entry or NULL if allocation failed. The values for the child reference is set to NULL for later assignment and the index entry is set to the largest possible volume.
Calls:	errorMessage() System: malloc()

```
Define index entry pointer, entry   /* pointer to index entry */
/*
 * Allocate memory
 */
Set entry = new IndexEntry
If entry == NULL Then
            Call errorMessage( Allocation failure in creating new index
               entry )
      Else
            Set entry.child          = NULL
            Set entry.key.lower.T    = MINIMUM_VALUE_OF_FLOAT
            Set entry.key.lower.X    = MINIMUM_VALUE_OF_FLOAT
            Set entry.key.lower.Y    = MINIMUM_VALUE_OF_FLOAT
            Set entry.key.lower.Z    = MINIMUM_VALUE_OF_FLOAT
            Set entry.key.upper.T    = MAXIMUM_VALUE_OF_FLOAT
            Set entry.key.upper.X    = MAXIMUM_VALUE_OF_FLOAT
            Set entry.key.upper.Y    = MAXIMUM_VALUE_OF_FLOAT
            Set entry.key.upper.Z    = MAXIMUM_VALUE_OF_FLOAT
      EndIf
      /*
       * Done
       */
      Return entry
```

Test:	

A.11.6 createIndexNode

Name:	createIndexNode
Input:	integer level, **level**
Output:	new index node, **node**
Return:	IndexNode pointer, or NULL (if allocation failed)
Description:	Routine creates new node using system allocation routine. Returns pointer to new node or NULL if allocation failed. The level of the new node is input and stored in new node and the list of index entries is specified as empty.
Calls:	errorMessage() System: malloc()

```
Get integer,       level            /* level of new node    */
Define index node pointer, node     /* pointer to new node  */
/*
 * Check for invalid node level
 */
If level < 0 Then
```

```
            Call errorMessage( Invalid level specified for new node )
            Set node = NULL
Else
            Set node = new IndexNode
            If node == NULL Then
                  Call errorMessage( Allocation failure in creating new
                     node )
            Else
                  Set node.level = level
            EndIf
EndIf
/*
 * Set entries to empty
 */
Set root.entries = EMPTY
/*
 * Done
 */
Return node
```

Test:

A.11.7 delete

Name:	delete
Input:	index root node, **root**
	search index key, **searchKey**
	search non-key values, **searchNonKeys**
Output:	updated index node, **root**
Return:	DELETE_SUCCESS, or
	DELETE_INVALID_KEY_SEARCH_VALUE,
	DELETE_INVALID_NON_KEY_SEARCH_VALUE,
Description:	
Calls:	deleteEntry()
	validKey()
	validNonKey()

```
Get index node, root
Get index key, searchKey
Get list of non-key values, searchNonKeys
Define boolean , adjustmentFlag
/*
 * Check key and non-key search values for validity if requested
 */
If validKey( searchKey ) == FALSE Then
            Call errorMessage( Invalid index key for DELETE )
            Return DELETE_INVALID_KEY_SEARCH_VALUE
Else If validNonKey( searchNonKeys ) == FALSE Then
            Call errorMessage( Invalid non-key list for DELETE )
            Return DELETE_INVALID_NON_KEY_ATTRIBUTE_CODE
EndIf

/*
 * Call deleteEntry routine for root node which will recursively
```

```
 * process the entire index.  Note that the adjustmentFlag is
 * passed to deleteEntry but it not needed for the root node since
 * adjustments would be made to the parent which the root does not
 * have.
 */
     Call deleteEntry( root, searchKey, searchNonKeys, adjustmentFlag
        )
/*
 * The final check is made to insure that the root node has more
 * than one child unless it is a leaf node.
 */
If root.level != LEAF Then
            Loop while number of entries on root <= 1
                  Set root = root.entries.child.node
            EndLoop
EndIf
/*
 * Done
 */
Return DELETE_SUCCESS
```

Test:

A.11.8 deleteDataObject

Name:	deleteDataObject	
Input:	data object to delete, **dataObject**	
Output:	none	
Return:	void	
Description:	Routine deletes given data object including all non-key character sequences.	
Calls:		
	System:	free()

```
Get DataObject, dataObject    /* data object to delete      */
/*
 * Delete the data object's non-key attribute values
 * Number of values is based on object type
 */
If dataObject.type == SMALL Then
            Loop for i = 0 to SMALL - 1
                  If dataObject.attributes[ NUMBER_OF_KEY_ATTRIBUTES +
                     i ] != NULL Then
                        Delete dataObject.attributes[
                           NUMBER_OF_KEY_ATTRIBUTES + i ]
                  EndIf
            EndLoop
Else If dataObject.type == MEDIUM Then
            Loop for i = 0 to MEDIUM - 1
                  If dataObject.attributes[ NUMBER_OF_KEY_ATTRIBUTES +
                     i ] != NULL Then
                        Delete dataObject.attributes[
                           NUMBER_OF_KEY_ATTRIBUTES + i ]
                  EndIf
            EndLoop
```

```
Else
            Loop for i = 0 to LARGE - 1
                  If dataObject.attributes[ NUMBER_OF_KEY_ATTRIBUTES +
                     i ] != NULL Then
                        Delete dataObject.attributes[
                           NUMBER_OF_KEY_ATTRIBUTES + i ]
                  EndIf
            EndLoop
EndIf
/*
 * Delete data object attributes array
 */
Delete dataObject.attributes
/*
 * Delete data object
 */
Delete dataObject
/*
 * Done
 */
Return
```

Test:

A.11.9 deleteEntry

Name:	deleteEntry
Input:	index node, **node** search index key, **searchKey** search non-key values, **searchNonKeys** boolean key adjustment flag, **adjustmentFlag**
Output:	updated index node, **node**
Return:	void
Description:	The routine is a recursive call to traverse the current index and delete all consistent data objects. The routine will also check for empty nodes and removes them as the index is traversed. The routine will treat leaf and non-leaf nodes differently. Non-leaf nodes have entries which reference other nodes. For each entry in a non-leaf node, the routine is recursive called on that entry. The recursive call will eventually descend to a leaf node. Upon return from the recursive call, the index branch referenced by the entry is in one of three states: (1) No data objects, entries, or nodes were removed in the branch. The branch is unaltered and no ajdustment to "upper" portions of the index is necessary. (2) One or more data objects, entries, or nodes were removed in the branch. The branch is altered and the index key value for the parent of the current node needs adjustment. However, the node referenced by the

branch, i.e., index entry, is NOT empty, and should not be removed. This means that the current entry is left intact, only its key is adjusted.

(3) One or more data objects, entries, or nodes were removed in the branch and the node referenced by the entry is now empty. The branch has been altered and the node, since its empty, should be removed along with the entry which references it.

When the current node is a leaf node, the data object each entry of the node references is checked for consistency with the input key and non-key search values and removed, along with the entry which references it, if appropriate. A flag is then set which tells the parent, i.e., the calling routine, that the node has been altered and a key adjustment is required.

Calls: consistentKey()
consistentNonKey()
deleteEntry()
keysUnion()

```
Get index node, node
Get index key, searchKey
Get list of non-key values, searchNonKeys
Get boolean , adjustmentFlag

/*
 * Set key adjustment flag to FALSE until adjustment is necessary
 * through a delete or return flag.
 */
Set adjustmentFlag = FALSE
/*
 * Treat leaf and non-leaf nodes differently.
 */
If node.level > LEAF Then
            /*
             * Current node isn't a leaf node, so descend branches
            until
             * leaf.  While descending, check input search index key
             * values only, since each entry will reference a node and
             * not a data object.
             */
            Loop for each entry tempEntry of node
                  If consistentKey( tempEntry.key, searchKey ) == TRUE
                     Then
                        /*
                         * Recursively call deleteEntry for child entry
                         * of current node
                         */
                        Call deleteEntry( tempEntry.child.node,
                           searchKey, searchNonKeys,
                           tempAdjustmentFlag)
                        /*
                         * After return from recursive deleteEntry
                        call,
                         * the index beneath this node is in one of
                         * three states: (1) No entries of
```

```
                    * tempEntry.child.node were removed, thus no
                    * key ajdustment is required, (2) some entries
                    * of the tempEntry.child.node were removed,
                   but
                    * some are left so need key adjustment, and
                   (3)
                    * all entries of the tempEntry.child.node were
                    * removed, so no key adjustment (adjust
                    * what?!?) and remove the entry (which also
                    * removes the node).
                    */
                   If tempEntry.child.node.entries == EMPTY Then
                         /*
                          * Remove the entry from the current
                         node,
                          * since the node it references is empty
                          * and no longer required.  Note that
                          * removing the entry should also remove
                          * whatever it references, i.e., typical
                          * index tree behavior.
                          */
                         Delete tempEntry removing it from node
                         Set adjustmentFlag = TRUE
                   Else If tempAdjustmentFlag == TRUE Then
                         /*
                          * One or more entries of the child node
                          * was removed so a key adjustment of the
                          * current entry is required.
                          */
                         Call tempEntry.key = keysUnion(
                            tempEntry.child.node.entries )
                         Set adjustmentFlag = TRUE
                   EndIf
             EndIf
       EndLoop
Else
       /*
        * Current node is a leaf node so each entry references a
        * data object.  Check input search key and non-key values
        * and remove all consistent data objects, setting
        * adjustment flag if removal occurs.
        */
       Loop for each entry tempEntry of node
             /*
              * Compare search key and stored data object key
              */
             If consistentKey( tempEntry.key, searchKey ) == TRUE
                Then
                   /*
                    * Set an upper bound for checking stored
                    * attributes to prevent out-of-bounds errors
                    */
                   Define integer upperBound = 0
                   If tempEntry.child.dataObject.type == SMALL
                   Then
                         Set upperBound = SIZE_OF_SMALL_ATTRIBUTE
```

```
                Else If tempEntry.child.dataObject.type ==
                  MEDIUM Then
                      Set upperBound = SIZE_OF_MEDIUM_ATTRIBUTE
                Else If tempEntry.child.dataObject.type ==
                  LARGE Then
                      Set upperBound = SIZE_OF_LARGE_ATTRIBUTE
                EndIf
                /*
                 * Check all non-key search values until done
                or
                 * until non consistent value found
                 */
                Define boolean acceptanceFlag = TRUE
                Loop for each DataAttribute nonKey in
                  searchNonKeys and while acceptanceFlag ==
                  TRUE
                      /*
                       * Only check search attributes which are
                       * relevant to data object
                       */
                      If nonKey.code < upperBound Then
                            /*
                             * Set acceptanceFlag to result of
                             * consistency check
                             */
                            Define integer attributeIndex =
                              nonKey.code -
                              NUM_OF_KEY_ATTRIBUTES
                            Set acceptanceFlag =
                              consistentNonKey(
                                nonKey.value,
                                tempEntry.attributes[
                            attributeIndex ] )
                      EndIf
                EndLoop
                /*
                 * If no non-key search values were
                 * inconsistent, delete the data object and
                 * index entry which references it.  Also, set
                 * ajdustment flag to tell calling process,
                 * i.e., parent node, to adjust its key.
                 */
                If acceptanceFlag == TRUE Then
                      Delete tempEntry removing from node
                      Set adjustmentFlag = TRUE
                EndIf
            EndIf
        EndLoop
EndIf
/*
 * Done
 */
Return DELETE_ENTRY_SUCCESS
```

Test:

A.11.10 deleteIndexEntry

Name:	deleteIndexEntry
Input:	index entry to delete, **entry** level where entry resides, **level**
Output:	none
Return:	void
Description:	Routine deletes input index entry by telling child, an index node or data object, to delete itself.
Calls:	deleteDataObject() deleteIndexNode() errorMessage() System: free()

```
Get index node,   entry        /* index entry to delete       */
/*
 * Delete the entry's child
 */
If level > LEAF Then
            /*
             * Entry's child is an index node
             */
            Call deleteIndexNode( entry.child.node )
Else If level == LEAF Then
            /*
             * Entry's child is a data object
             */
            Call deleteDataObject( entry.child.dataObject )
Else
            /*
             * Don't know what entry's child is, because the level is
             * negative which is undefined.
             */
            Call errorMessage( "WARNING: unknown level specified for
               entry deletion, possible memory leak )
EndIf
/*
 * Delete the entry
 */
Delete entry
/*
 * Done
 */
Return
```

Test:

A.11.11 deleteIndexNode

Name:	deleteIndexNode
Input:	node to delete, **node**

Output:	none
Return:	void
Description:	Routine deletes input node. Recursively descends all children of node to allow deletion of branches.
Calls:	deleteIndexEntry()
	System: free()

```
Get index node,   node        /* node to delete */
/*
 * Delete the index entries which reside on node
 */
Loop for each entry, entry, in node.entries
            /*
             * Delete the entry
             */
            Call deleteIndexEntry( node.level, entry )
EndLoop
/*
 * Delete the node
 */
Delete node
/*
 * Done
 */
Return
```

Test:

A.11.12 insert

Name:	insert
Input:	index (root node), **root** new index entry, **entry** integer fan value, **fan**
Output:	Updated index, **root**
Return:	INSERT_SUCCESS, or INSERT_INSERT_ENTRY_FAILURE_FATAL INSERT_INSERT_ENTRY_FAILURE_NON_FATAL INSERT_ALLOCATION_FAILURE
Description:	Place data object into the index with specified fan returning the index via the root node. The insert method descends the tree until the specified level is reached. Note that the leaf level is zero and the level increases as the tree ascends, i.e., the root level is always greater than or equal to the leaf level. The branch or node chosen for descent is determined by comparing the penalty for all possible branches and the new entry. The branch which has the smallest or minimum penalty is chosen. Once the specified level is reached, a node is chosen on that level which yields the minimum penalty and the entry is placed on that node. Placement of the entry onto the node may exceed the specified fan which causes the node to split. The node split separates the union of the old entries of the node and the new entry

into two groups. One group is placed back onto the old node, and the other group is placed on a new node created for that purpose. The new node is then placed onto the parent of the old node. The may cause the parent to split and an identical splitting process is carried out on the parent, which may cause its parent to split, etc. This splitting may ascend to the root node, which by definition has no parent and so is a special case. When the root node is split, a new root is created which "grows" the tree. The old root and the node split off of the old root are then placed onto the new root, and the new root is returned as the updated index.

The *insert* routine creates a new entry for the data object. Then, the routine places the new entry into the index using the *insertEntry* subroutine using the input root node as the start of the tree descent. If the root was split, the tree "grows" by creating a new root and placing the old root and the split entry onto the new root and returning the new root as the updated index. The primary difference between *insert* and *insertEntry* is that *insert* is used for the special case of the root node splitting while *insertEntry* is for every other node all of which have parents for split entry placements. A modified *insertEntry* which checks for root splitting is possible and would have the advantage of eliminating an "extra" subroutine since there would be no need for *insert*. However, the *insert* routine is used for this implementation because the modified *insertEntry* would check each node for root splitting, which isn't necessary, and the additional clarity of placing the special case of root splitting into a separate routine

Calls: createIndexEntry()
errorMessage()
insertEntry()
keysUnion()

```
Get index node ptr, root               /* root node of index    */
Get data object ptr, dataObject        /* new object to place   */
Get integer fan, fan                   /* fan of index tree     */
Define index entry ptr, entry          /* entry of new object   */
Define index entry ptr, splitEntry     /* entry after split     */

Assert that 0 < fan < MINIMUM_FAN_SIZE
/*
 * Create index entry.  Set child of entry to reference new data
 * object and set index key of entry to appropriate values of data
 * object.
 */
Call entry = createIndexEntry( LEAF )
If entry == NULL Then
            /*
             * Couldn't allocate memory for new index entry for new
            data
             * object.
             */
            Return INSERT_ALLOCATION_FAILURE
EndIf
```

```
Set entry.child.dataObject = dataObject
Set entry.key = dataObject key values
/*
 * Place new entry on root node, splitting if necessary.
 */
Call error = insertEntry( root, entry, LEAF, fan, splitEntry )
If error == INSERT_ENTRY_SUCCESS Then
            /*
             * Check for split and grow tree if necessary.  If a split
             * occurred, then the root node was split.  Since the root
             * node has no parent, a split root node requires that a
            new
             * root be created and the tree to grow.
             */
            If splitEntry != NULL Then
                  /*
                   * Need new root, i.e., grow the tree
                   */
                  Define index node, newRoot
                  Call newRoot = createIndexNode( root.level + 1 )
                  /*
                   * Couldn't allocate new root node.  This is a fatal
                   * error since a child node was split and the index
                   * altered.  Because the current implementation does
                   * not employ an "undo" command, the result is a
                  fatal
                   * error since the current state of the index is not
                   * known.
                   */
                  If newRoot == NULL Then
                        Call errorMessage( allocation failure )
                        Return INSERT_ALLOCATION_FAILURE
                  EndIf
                  /*
                   * Create new entry for old root
                   */
                  Define index entry, newEntry
                  Call newEntry = createIndexEntry()
                  /*
                   * Couldn't allocate new entry for old root node.
                   * This is a fatal error, ... (see above)
                   */
                  If newEntry == NULL Then
                        Call errorMessage( allocation failure )
                        Return INSERT_ALLOCATION_FAILURE
                  EndIf
                  /*
                   * Setup newEntry to reference the old root
                   */
                  Set newEntry.child.node = root
                  Set newEntry.key        = keysUnion( root.entries )
                  /*
                   * Place newEntry and splitEntry onto newRoot
                   */
                  Place newEntry on newRoot
                  Place splitEntry on newRoot
                  /*
```

```
                    * Update root for return
                    */
                   Set root = newRoot
            EndIf
/*
 * A chooseEntry failure occurred for a child.  The
 * chooseEntry routine is only called while descending the
 * tree, so an error of this type means that the current
 * index has not been altered, and so the error is non-
 * fatal.
 */
Else If error == INSERT_ENTRY_CHOOSE_ENTRY_FAILURE Then
            Call errorMessage( failed to place new entry )
            Return INSERT_INSERT_ENTRY_FAILURE_NON_FATAL
/*
 * A fatal error occurred while splitting on some child of the
 * root node.  The current index has been altered prior to the
 * error and no recovery is possible.
 */
Else If error == INSERT_ENTRY_SPLIT_NODE_FATAL Then
            Call errorMessage( failed to place new entry )
            Return INSERT_INSERT_ENTRY_FAILURE_FATAL
/*
 * A non-fatal error occurred while splitting a leaf node.  The
 * current index has not been altered prior to the error.
 */
Else If error == INSERT_ENTRY_SPLIT_NODE_NON_FATAL Then
            Call errorMessage( failed to place new entry )
            Return INSERT_INSERT_ENTRY_FAILURE_NON_FATAL
EndIf
/*
 * Done
 */
Return INSERT_SUCCESS
```

Test:

A.11.13 insertEntry

Name:	insertEntry
Input:	node to place new entry, **node** new index entry, **entry** level to place entry, **level** integer fan value, **fan**
Output:	possible new entry from splitting, **splitEntry**
Return:	INSERT_ENTRY_SUCCESS, or INSERT_ENTRY_CHOOSE_ENTRY_FAILURE INSERT_ENTRY_SPLIT_NODE_FATAL INSERT_ENTRY_SPLIT_NODE_NON_FATAL
Description:	Inserts entry onto provided node at specified level. If the current node is not at the specified level, i.e., above, the method *chooseEntry* is used to determine best entry of current node for insertion and then the new entry is inserted onto the child of the chosen entry using recursive call. If the

	current node is at the specified level, the new entry is placed on node splitting if necessary via the *splitNode* method. The output is the possible new sibling or split entry which is passed to the calling routine for parent insertion. If no splitting occurs, the split entry value is set to NULL.
Calls:	chooseEntry ( ) errorMessage() insertEntry() keysUnion() splitNode( )

```
Get index node, node            /* node to place entry on      */
Get index entry, entry          /* new entry to place          */
Get integer, level              /* level to place entry        */
Get integer fan, fan            /* fan or order of index tree  */
Get index entry, splitEntry     /* entry after split           */
Define integer, error           /* error flag                  */

Assert that 0 < fan < MINIMUM_FAN_SIZE
Assert that level >= LEAF

/*
 * If the current node is not at the specified level, then choose
 * branch and descend unitl correct level is reached.
 */
If node.level > level Then
            /*
             * Choose index entry on node to place entry. Note that the
             * only way for chooseEntry to fail is for node to be
            empty.
             */
            Define index entry, chosen
            Set chosen = chooseEntry( node, entry )
            If chosen == NULL Then
                  /*
                   * Can't choose an entry on node because node is
                   * empty, which means tree is unbalanced.
                   */
                  Call errorMessage( unbalanced tree )
                  Return INSERT_ENTRY_CHOOSE_ENTRY_FAILURE
            EndIf
            /*
             * Descend tree through chosen branch until level is
            reached
             */
            Call error = insertEntry( chosen.child.node, entry, level,
               fan, splitEntry )
            /*
             * A successful insertEntry call causes a key adjustment
             * and a check to see if splitting occurred.
             */
            If error == INSERT_ENTRY_SUCCESS Then
                  /*
                   * Adjust key after insertion
                   */
                  Set chosen.key = keysUnion(chosen.child.node.entries)
                  /*
```

```
 * Check for splitting
 */
If splitEntry != NULL Then
      /*
       * Child was split and splitEntry is valid.
       * Check to see if current node is full.  If
       * not, place the split entry onto the current
       * node.  Otherwise, split current node.
       */
      If number of node.entries < fan Then
            /*
             * There is room for entry on this node
             */
            Place splitEntry on node
            /*
             * No splitting
             */
            Set splitEntry = NULL
      Else
            Define temp index entry, tempEntry
            /*
             * Place new entry by splitting node.
             * Note that the splitEntry is being
             * placed on node which will create a new
             * node/entry sibling called tempEntry.
             */
            Call error = splitNode( node, fan,
               splitEntry, tempEntry )
            /*
             * Check the return code for a non-
             * successful split.  Any non-successful
             * split will cause a fatal error to
            occur
             * since the current implementation does
             * not attempt an "undo" mechanism, i.e.,
             * saving the original branch and
             * restoring the index tree to its
             * previous state.  The only case where a
             * splitNode failure is non-fatal, is
            when
             * the splitting occurred at the LEAF
             * level which means the index is still
            in
             * its original state.
             */
            If error != SPLIT_NODE_SUCCESS Then
                  Return
                  INSERT_ENTRY_SPLIT_NODE_FATAL
            EndIf
            /*
             * Set splitEntry to tempEntry value.
            The
             * old value for splitEntry was placed
             * onto node above.  The value of
             * tempEntry needs to be placed on the
             * parent of node which is accomplished
            by
```

```
                         * setting the value of splitEntry to the
                         * correct value.
                         */
                        Set splitEntry = tempEntry
                EndIf
        EndIf
    /*
     * A chooseEntry failure occurred for a child.  The
     * chooseEntry routine is only called while descending the
     * tree, so an error of this type means that the current
     * index has not been altered, and so the error is non-
     * fatal.
     */
    Else If error == INSERT_ENTRY_CHOOSE_ENTRY_FAILURE Then
        /*
         * Propogate error to calling process
         */
        Return INSERT_ENTRY_CHOOSE_ENTRY_FAILURE
    /*
     * A fatal error occurred during a node split.  The error
is
     * fatal because the index was altered prior to the error
     * occuring.  Thus, the state of the index is not known
     * (balanced,etc.).
     */
    Else If error == INSERT_ENTRY_SPLIT_NODE_FATAL Then
        Return INSERT_ENTRY_SPLIT_NODE_FATAL
    /*
     * A non-fatal error occurred during a node split.  This
     * error is non-fatal because the split error occurred
     * before the index was altered, i.e., during a leaf node
     * split.
     */
    Else If error == INSERT_ENTRY_SPLIT_NODE_NON_FATAL Then
        Return INSERT_ENTRY_SPLIT_NODE_NON_FATAL
    EndIf
Else
    /*
     * Install entry on node, splitting if necessary
     */
    If number of node.entries < fan Then
        /*
         * There is room for entry on this node
         */
        Place entry on node
        /*
         * No splitting
         */
        Set splitEntry = NULL
    Else
        /*
         * Place new entry by splitting node
         */
        Call error = splitNode( node, fan, entry, splitEntry
)
        If error == SPLIT_NODE_ALLOCATION_FAILURE Then
            /*
```

```
                                  * Allocation error occurred when the node
                                  * attempted to split for new entry.  This is a
                                  * FATAL error, unless the node was a LEAF node
                                  * which means a splitNode failure has not yet
                                  * effected the index tree.  If the node which
                                  * failed to split is a LEAF node, return a
                                 non-
                                  * fatal error code.
                                  */
                                 Call errorMessage( failure to split node in
                                    insertEntry )
                                 If node.level == LEAF Then
                                       Return INSERT_ENTRY_SPLIT_NODE_NON_FATAL
                                 Else
                                       Return INSERT_ENTRY_SPLIT_NODE_FATAL
                                 EndIf
                           EndIf
                     EndIf
EndIf
/*
 * Done
 */
Return INSERT_ENTRY_SUCCESS
```

Test:

A.11.14 partitionEntries

Name:	partitionEntries
Input:	input list of entries, **I** integer fan value, **fan**
Output:	output list of entries, **A** output list of entries, **B**
Return:	void
Description:	Separate input list of index entries into two groups. The method used for partitioning the entries is extremely implementation dependent. The basic idea is to set-up the two output index entry lists to have minimal bounding hyper-cubes which will improve later queries on the index since fewer branches of the index will need to be traversed to satisfy the query command. However, the method itself is probably the most computationally expensive of the insertion subroutines, because multiple loops through the index entry lists and penalty calculations are required for true "minimal" bounding hyper-cubes to be determined. If multiple branch searches is not prohibitive, i.e., a parallel search is possible or query response is not time consuming relative to an insert operation, then the partition subroutine can use a sub-minimal approach. In fact, the partition can simply split the input list into two equal groups ignoring the bounding hyper-cubes completely. The effect will be to cause new traversals of the index to descend multiple branches, but this trade-off may be acceptable for a given implementation.

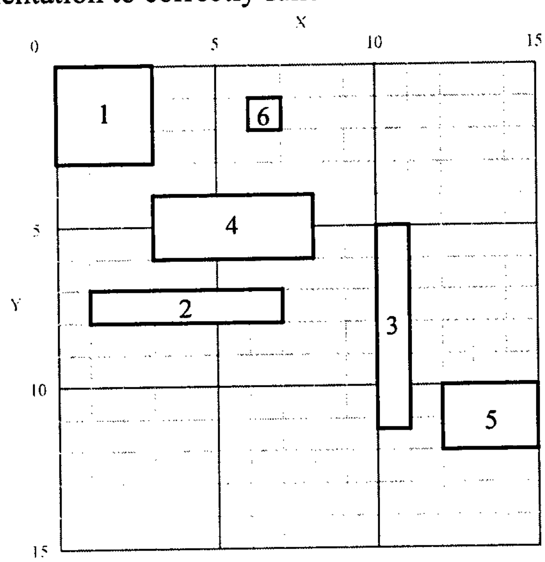
	The partitioning takes place in two stages. The first stage finds the two entries within the input list which produce the "largest" index key, or the two entries whose union has the greatest hyper-cube volume among all possible combinations of the input list. Each of the entries which form this "worst" pair is used as the seeds, or first members, of the partitioned groups, **A** and **B**, and are "removed" from the input list. The second stage assigns the remaining entries of the input list **I** to one of the partitioned groups based on which group seed yields the smaller penalty. A smaller penalty indicates an "attraction" to that group and will yield small total entry index keys which in turn improves query response. The second stage has a special case where one of the two partitioned groups is full. In this case, all of the remaining entries are assigned to the other group. On entrance, the list **I** should have at least MINIMUM_FAN_SIZE entries and no more than 2*fan entries, and the output lists **A** and **B** should be empty(NULL). On exit, the input list **I** is empty (NULL), and the output lists **A** and **B** have the partitioned entries. Each list is ready for insertion into an IndexNode.
Calls:	errorMessage() penalty()

```
Get list of index entries, I   /* input list to partition        */
Get list of index entries, A   /* 1st group of I                 */
Get list of index entries, B   /* 2nd group of I                 */
Get integer fan value, fan     /* fan of index                   */

Assert that fan < MINIMUM_FAN_SIZE
Assert that size of list I is < MINIMUM_FAN_SIZE
Assert that size of list I > 2 * fan
/*
 * Stage 1
 *
 * Find "worst" combination of all entries in I.  The worst
 * combination is the one which produces the largest key union, or
 * the largest bounding hyper-cube.  The two entries which form
 * this worst combo will be the first entries into the two groups.
 * The method used to find the worst combo is a straight forward
 * enumeration of all possible combinations.  Note that only
 * forward combinations are necessary, i.e., the volume of the
 * union(A,B) is the same as the volume of the union(B,A).  The
 * first candidate pair for the worst case are chosen as the first
 * and second entries of the input list, I.
 */
Define index entry, seedA      /* first entry in group A        */
Define index entry, seedB      /* first entry in group B        */
Set seedA = first entry of I
Set seedB = second entry of I
/*
 * A double loop through the input list, I, is used to find all
 * combinations.
 */
Loop for each index entry, entry1, in I
            Loop for each index entry, entry2, in I after entry1
```

```
                    /*
                     * If this combination produces a worse pair, then
                     * replace old candidates with new pair.
                     */
                    If volume of union( entry1, entry2 ) > volume of
                       union( seedA, seedB ) Then
                          Set seedA            = entry1
                          Set seedB            = entry2
                    EndIf
              EndLoop
EndLoop
/*
 * The entries, A and B, now point to the first entries
 * into the two groups which are forming during the partition.
 */
Add seedA to A and remove from I
Add seedB to B and remove from I
/*
 * Stage 2
 *
 * Assign all remaining entries of I to each group based on
 * penalty.  The current implementation finds the penalty of the
 * entry with the first entries into the groups.  Other methods
 * are possible, including using the penalty of the current
 * group.  Once an index entry as been assigned to a group, A or
 * B, remove that entry from the input list, I.
 */
Loop for each index entry, entry, in I
            /*
             * If either A or B is full, then place rest of entries on
             * list which is not full.
             */
            If size of A >= fan Then
                  Add entry to B
                  Remove entry from I
            Else If size of B >= fan Then
                  Add entry to A
                  Remove entry from I
            Else
                  /*
                   * Neither A or B is full, so assign entry to group
                   * based on penalty.
                   */
                  If volume of union( entry, seedA ) > volume of union(
                     entry, seedB ) Then
                        Add entry to A
                  Else
                        Add entry to B
                  EndIf
                  Remove entry from I
            EndIf
EndLoop
/*
 * Done
 */
Return
```

Test:	The test of the partitionEntries() routine will consist of a known input/output test case consisting of a group of pseudo-2D objects. The objects for the case are shown in the following figure with an underlying grid to facilitate the test description. The objects are shown as two-dimensional, using X and Y, and the last two dimensions are specified to have a "thickness" of unity. This allows the four-dimensional R-Tree implementation to correctly function.



The index keys (bounding-box) for each object is given in the following table. Note that the "thickness" of the T and Z dimensions are unity.

		Object 1	2	3	4	5	6
Lower Point	**T**	0	0	0	0	0	0
	X	0	1	10	3	12	6
	Y	0	7	5	4	10	1
	Z	0	0	0	0	0	0
Upper Point	**T**	1	1	1	1	1	1
	X	3	7	11	8	15	7
	Y	3	8	11	6	12	2
	Z	1	1	1	1	1	1

The first stage of partitioning finds the "worst" pair, the pair with the largest index key union, as the seeds for the two new partitioned groups. The following table lists the penalties for all possible pairs for the test case.

Object	Object	Volume
1	2	56

1	3	121
1	4	48
1	5	180
1	6	21
2	3	60
2	4	28
2	5	70
2	6	42
3	4	56
3	5	35
3	6	50
4	5	96
4	6	25
5	6	99

The "worst" case is the pair of Object 1 and Object 5 and each object forms the seed for, or is the first member of, the two new partitions. The next stage of partitioning adds the rest of the objects to each group based on which group seed will yield the lowest volume. The following table shows each of the remaining objects and their volume for both Object 1/Group A and Object 5/Group B with the group assignment highlighted in the table.

Object	Penalty(1, Object)	Penalty(5, Object)
2	47	64
3	112	29
4	39	90
6	12	93

A successful test of the paritionEntries() routine will produce the correct values of the volumes as well as a correct parition of the groups where the first group will have objects 1, 2, and 4 and the second group will objects 3 and 5.

A secondary test would be to set the fan size to less than four. This will force the last entry, Object 6, to be placed in group B rather than group A.

A.11.15 <u>penalty</u>

Name:	penalty
Input:	index entry, **A**
	index entry, **B**
Output:	float value of penalty, **penalty**
Return:	float value, **penalty**

Description:	Calculate and return the penalty for the two input index entries. The penalty for the index is defined as the "change in area" proposed by Guttman in the original 1984 paper. Given two index entries, **A** and **B**, the penalty is defined as penalty = volume(union(**A, B**)) - volume(**A**) Note that the penalty routine is not communative, i.e., the penalty(**A, B**) is not necessarily the same as the penalty of (**B, A**)
Calls:	
	System:

```
Get index entry,  A              /* input index entry     */
Get index entry,  B              /* input index entry     */
Get float,        penalty        /* calculated penalty    */
Define index key, key            /* index key of union    */
/*
 * Find union of A and B keys
 */
Set key = union of A.key and B.key
/*
 * Determine theh increase in volume which is the penalty
 */
     Set penalty = volume of key - volume of A.key
/*
 * Done
 */
Return penalty
```

Test:	

A.11.16 query

Name:	query
Input:	index (root node), **node** search index key, **searchKey** search non-key values, **searchNonKeys** check validity flag, **checkValidity**
Output:	List of index entries, **solutionSet**
Return:	QUERY_SUCCESS QUERY_INVALID_KEY_SEARCH_VALUE QUERY_INVALID_NON_KEY_ATTRIBUTE_CODE
Description:	Searches index and returns list of index entries which are consistent with the search keys. The routine first uses the R-Tree index to find individual leaf nodes which point to data object's which have index keys which are consistent with the search key. The found data object's non-key attributes are then compared with the input list of non-key search values for consistency. The input list of non-key search values consist of the attribute code and a character sequence. Two utility subroutines are used to check the validity of the input search values. The input is checked only when the input flag is set to TRUE which prevents multiple checks of the same data since this routine is applied recursively. Note that the index is

	never altered by a query which means no error is fatal. Two utility subroutines are used to determine consistency which is intersection for the DIS application. The utility subroutines are used to separate the specific hyper-cube dimension and character string from the general R-Tree algorithm.
Calls:	consistentKey() consistentNonKey() validKey() validNonKey()

```
Get index node, node                          /* sub-tree node  */
Get index key, searchKey                      /* key for search */
Get list of non-key values, searchNonKeys     /* non-key search */
Get integer , checkValidity                   /* validity flag  */
Define index entry list, solutionSet          /* solution list  */
Set solutionSet = EMPTY

/*
 * Check key and non-key search values for validity if requested
 */
If checkValidity == TRUE Then
            If validKey( searchKey ) == FALSE Then
                  Call errorMessage( Invalid key search )
                  Return QUERY_INVALID_KEY_SEARCH_VALUE
            Else If validNonKey( searchNonKeys ) == FALSE Then
                  Call errorMessage( Invalid non-key search )
                  Return QUERY_INVALID_NON_KEY_ATTRIBUTE_CODE
            EndIf
EndIf
/*
 * Search on key attributes first.  All data objects are
 * referenced from the leaf level, so any query which is above
 * that level should check each branch, descending if necessary.
 */
If node.level > LEAF Then
            /*
             * Descend tree until leaf.  Note that the level decreases
             * as the tree is descended with level = LEAF = 0 being a
             * leaf node.
             */
            Loop for each entry tempEntry of node
                  If consistentKey( tempEntry.key, searchKey ) == TRUE
                     Then
                        Define temporary solution set, tempSolutionSet
                        /*
                         * Query on all consistent children w/o
                        checking
                         * input since only the root or top node should
                         * check only once.
                         */
                        Call query( tempEntry.child.node, searchKey,
                           searchNonKeys, FALSE, tempSolutionSet )
                        Add tempSolutionSet to solutionSet
                  EndIf
            EndLoop
Else
```

```
Loop for each entry tempEntry of node
      /*
       * Compare search key and stored data object key
       */
      If consistentKey( tempEntry.key, searchKey ) == TRUE
         Then
            /*
             * Set an upper bound for checking stored
             * attributes to prevent out-of-bounds errors
             */
            Define integer upperBound = 0
            If tempEntry.child.dataObject.type == SMALL
            Then
                  Set upperBound = SIZE_OF_SMALL_ATTRIBUTE
            Else If tempEntry.child.dataObject.type ==
               MEDIUM Then
                  Set upperBound = SIZE_OF_MEDIUM_ATTRIBUTE
            Else If tempEntry.child.dataObject.type ==
               LARGE Then
                  Set upperBound = SIZE_OF_LARGE_ATTRIBUTE
            EndIf
            /*
             * Check all non-key search values until done
            or
             * until non consistent value found
             */
            Define integer acceptanceFlag = TRUE
            Loop for each DataAttribute nonKey in
               searchNonKeys and while acceptanceFlag ==
               TRUE
                  /*
                   * Only check search attributes which are
                   * relevant to data object
                   */
                  If nonKey.code < upperBound Then
                        /*
                         * Set acceptanceFlag to result of
                         * consistency check
                         */
                        Define integer attributeIndex =
                           nonKey.code -
                           NUM_OF_KEY_ATTRIBUTES
                        Set acceptanceFlag =
                           consistentNonKey( nonKey.value,
                           tempEntry.attributes[
                           attributeIndex ] )
                  EndIf
            EndLoop
            /*
             * If no non-key search values were
             * inconsistent, add data object to solution
             * set.  At this point, the data object could
             * also be placed in an output queue rather
            than
             * adding it to a list.
             */
            If acceptanceFlag == TRUE Then
```

```
                              Add tempEntry to solutionSet
                          EndIf
                      EndIf
                  EndLoop
EndIf
/*
 * Done
 */
Return QUERY_SUCCESS
```

Test:

A.11.17 splitNode

Name:	splitNode
Input:	node to split, **nodeToSplit** fan or order of index tree, **fan** new entry which caused split, **entry**
Output:	entry of new node after splitting, **splitEntry**
Return:	SPLIT_NODE_SUCCESS, or SPLIT_NODE_ALLOCATION_FAILURE
Description:	This routine splits an index node. An index node needs to split whenever a new index entry is added to the node and the node is full, i.e., the current number of entries residing on the node is equal to the fan or order of the index tree. A node split consists of dividing the current entries of a node and the new entry into two groups via a *partitionEntries* routine. One group is placed onto the node and the other group is placed onto a new node created for that purpose. A new index entry is created for the new node and is returned as the splitEntry in the output. Since both a new index node and a new index entry are created during the splitting process, a memory allocation failure is possible during the execution of this subroutine. The allocation failure is fatal for most uses of the method, but in certain cases, splitting a leaf node for instance, a recovery is possible at a higher level. For this reason, the *splitNode* routine will "clean-up" before returning, i.e., deallocating memory, etc.
Calls:	createIndexEntry() createIndexNode() deleteIndexNode() errorMessage() keysUnion() partitionEntries() System:

```
Get index node, nodeToSplit    /* node to split              */
Get integer, fan               /* fan or order of index tree */
Get index entry, entry         /* entry to add to node       */
Get index entry, splitEntry    /* entry of new node          */

Assert that fan < MINIMUM_FAN_SIZE
/*
```

```
 * Create new node for partitioning.  The new node is a
 * "sibling" of the input node, so its level is the same.
 */
Call tempNode = createIndexNode( nodeToSplit.level )
If tempNode == NULL Then
            Call errorMessage( allocation failure )
            Return SPLIT_NODE_ALLOCATION_FAILURE
EndIf
/*
 * Create new entry which references the newly created node.
 */
Call splitEntry = createIndexEntry();
If splitEntry == NULL Then
            /*
             * Free tempNode memory
             */
            Call deleteIndexNode( tempNode )
            /*
             * Alert user of error and return error code
             */
            Call errorMessage( allocation failure )
            Return SPLIT_NODE_ALLOCATION_FAILURE
Else
            /*
             * The child of the split entry is the new node
             */
            Set splitEntry.child = tempNode
/*
 * Create a list which is the current entries of the node to split
 * and the entry which caused the split to occur.
 */
Define list of index entries, listOfEntries
Set listOfEntries = nodeToSplit.entries + entry
/*
 * Partition entries onto old nodeToSplit and new temporary node
 */
      Call partitionEntries( listOfEntries, fan, nodeToSplit.entries,
         tempNode.entries )
      /*
       * Adjust key of splitEntry for new node. Note that the key of
         the entry for nodeToSplit is adjusted in the same process that
         called splitNode.
       */
Set splitEntry.key = keysUnion( tempNode.entries )
/*
 * Done
 */
Return SPLIT_NODE_SUCCESS
```

Test:

A.11.18 validKey

Name:	validKey
Input:	index key, **key**

Output:	flag indicating whether key is valid
Return:	TRUE or FALSE
Description:	Returns boolean value indicating if the index key is valid, i.e., the lower point is actually lower than the upper point, etc.
Calls:	
	System:

```
Get index key, key       /* index key for search        */
/*
 * Check hyper-points
 */
If key.lower.T > key.upper.T Then
            Call errorMessage( Lower T > upper T )
            Return FALSE
Else If key.lower.X > key.upper.X Then
            Call errorMessage( Lower X > upper X )
            Return FALSE
Else If key.lower.Y > key.upper.Y Then
            Call errorMessage( Lower Y > upper Y )
            Return FALSE
Else If key.lower.Z > key.upper.Z Then
            Call errorMessage( Lower Z > upper Z )
            Return FALSE
EndIf
/*
 * Done
 */
Return TRUE
```

Test:	

A.11.19 validNonKey

Name:	validNonKey
Input:	list of non-key values, **nonKeys**
Output:	flag indicating valid attribute codes and strings
Return:	TRUE or FALSE
Description:	Returns boolean value indicating if non-key attribute list is valid. The condition for validity is that all of the data attribute codes within the list must lie within both the min/max of all attribute codes and are not key attribute codes, and that the character sequence is non-NULL.
Calls:	errorMessage()
	System:

```
/* non-key values */
Get list of non-key values, DataAttribute nonKeys
/*
 * Check non-key search values for valid attribute codes
 */
Loop for each DataAttribute nonKey in nonKeys
            /*
             * Check that the attribute code lies with min/max range
             */
```

```
            If nonKey.code < MIN_ATTRIBUTE_CODE || nonKey.code >=
               MAX_ATTRIBUTE_CODE Then
                  Call errorMessage( Out-of-range non-key
                  attributecode)
                  Return FALSE
            /*
             * Check that the attribute code is not for a key attribute
             */
            Else If nonKey.code < NUM_OF_KEY_ATTRIBUTES Then
                  Call errorMessage( Invalid non-key attribute code )
                  Return FALSE
            /*
             * Check that the attribute value (char string) is non-NULL
             */
            Else If nonKey.value == NULL Then
                  Call errorMessage( Empty string for non-keyattribute
                  )
                  Return FALSE
            EndIf
EndLoop
/*
 * Done
 */
Return TRUE
```

Test:

A.12 INPUT & OUTPUT

A.12.1 closeFiles

Name:	closeFiles
Input:	input FILE pointer, **inputFile**
	output FILE pointer, **outputFile**
	metrics FILE pointer, **metricsFile**
Output:	none
Return:	void
Description:	Closes the three files used during application.
Calls:	errorMessage()
	System: fclose()

```
Get input FILE pointer,       inputFile
Get output FILE pointer,      outputFile
Get metrics FILE pointer,     metricsFile
/*
 * Input File
 */
Call error = fclose( inputFile )
If error != ZERO Then
            Call errorMessage( Error closing inputFile )
EndIf
/*
 * Output File
 */
```

```
      Call error = fclose( outputFile )
If error != ZERO Then
            Call errorMessage( Error closing outputFile )
EndIf
/*
 * Metric File
 */
      Call error = fclose( metricsFile )
If error != ZERO Then
            Call errorMessage( Error closing metricsFile )
EndIf
/*
 * Done
 */
Return CLOSE_FILES_SUCCESS
```

Test:

A.12.2 flushOutputBuffer

Name:	flushOutputBuffer
Input:	output FILE ptr, **outputFile**
Output:	none
Return:	FLUSH_OUTPUT_BUFFER_SUCCESS, or FLUSH_OUTPUT_BUFFER_FAILURE
Description:	
Calls:	errorMessage() System: fopen()

```
Get output FILE ptr,      outputFile
/*
 * Done
 */
Return FLUSH_OUTPUT_BUFFER_SUCCESS
```

Test:

A.12.3 getDeleteCommand

Name:	getDeleteCommand
Input:	FILE pointer, **file**
Output:	index key, **searchKey** list of non-key values as data attributes, **searchNonKey**
Return:	GET_DELETE_SUCCESS or GET_DELETE_IO_ERROR GET_DELETE_INVALID_CODE_ERROR GET_DELETE_EARLY_EOF_OR_EOL_ERROR GET_DELETE_ALLOCATION_ERROR
Description:	Reads delete command from input stream via FILE pointer. Assumes current stream pointer is correct, and returns file pointer open and current position immediately after command just read. The file pointer is

expected to be at the beginning of the Delete command immediately after the command code. The Delete command consists of a list of attribute code and value pairs. An error occurs if any attribute code does not have an accompanying attribute value. For any error during read (missing pair, etc.), the routine will leave the current values of the command attribute list intact and clear the FILE pointer to the end of the current line.

A Delete command is made up of a list of attribute code and value pairs. The total number of pairs can range from zero to the MAX_ATTRIBUTE_CODE. Each command is carriage return delimited, i.e., one line per command. So, to read the Delete command, read until the line and store the attribute code/value pairs as we go along.

It is possible to have a delete command return an empty attribute list without producing an error. Since missing attributes are defaulted to wild-card values, this type of delete would logically remove the entire database. This "database" delete is a valid delete command as defined by the specification.

Calls: clearLine()
errorMessage()
System: scanf()
strcpy()

```
Get FILE pointer, file                        /* file to read          */
Get index key, searchkey                      /* key part of command   */
Get non-key DataAttribute list, searchNonKey     /* non-key part */
                                                 /* of command   */

/*
 * Set searchKey to wild-card values (max or min)
 */
Set searchKey.lower.T   = MINIMUM_VALUE_OF_FLOAT
Set searchKey.lower.X   = MINIMUM_VALUE_OF_FLOAT
Set searchKey.lower.Y   = MINIMUM_VALUE_OF_FLOAT
Set searchKey.lower.Z   = MINIMUM_VALUE_OF_FLOAT

Set searchKey.upper.T   = MAXIMUM_VALUE_OF_FLOAT
Set searchKey.upper.X   = MAXIMUM_VALUE_OF_FLOAT
Set searchKey.upper.Y   = MAXIMUM_VALUE_OF_FLOAT
Set searchKey.upper.Z   = MAXIMUM_VALUE_OF_FLOAT
/*
 * Set non-key attribute list to empty which clears out anything
 * which was left over from previous reads or from initialization.
 */
Set searchNonKey = EMPTY
/*
 * Check assumptions for attribute code values
 */
Assert that NUM_OF_KEY_ATTRIBUTES < total number of attributes
      Assert that every key attribute code < every non-key attribute
          code
/*
 * Read Delete command
 *
```

```
 * A Delete command is made up of a list of attribute code and
 * value pairs.  The total number of pairs can range from zero to
 * the MAX_ATTRIBUTE_CODE.  Each command is carriage return
 * delimited, i.e., one line per command. So, to read the Delete
 * command, read until the line and store the attribute code/value
 * pairs as we go along.
 */
While not at end-of-file or end-of-line
            Define integer attributeCode
            Read attributeCode
            /*
             * Check for I/O error (low-level fault in stream)
             */
            If I/O error during read Then
                  Call errorMessage( error in read )
                  Return GET_DELETE_IO_ERROR
            /*
             * Check for end-of-file(EOF) or end-of-line(EOL)
             */
            Else If end-of-file || end-of-line Then
                  /*
                   * Normal termination of Delete read
                   */
                  Return GET_DELETE_SUCCESS
            /*
             * Check for invalid attribute code
             */
            Else If attributeCode < MIN_ATTRIBUTE_CODE or attributeCode
               > MAX_ATTRIBUTE_CODE Then
                  Call errorMessage( WARNING: Invalid attribute code )
                  Call clearLine()
                  Return GET_DELETE_INVALID_CODE_ERROR
            /*
             * Check code for key attribute value
             */
            Else If attributeCode < NUM_OF_KEY_ATTRIBUTES Then
                  /*
                   * Read key attribute value
                   */
                  Define Float value
                  Read value
                  /*
                   * Check for I/O error (low-level fault in stream)
                   */
                  If I/O error in read Then
                        Call errorMessage( error in read )
                        Return GET_DELETE_IO_ERROR
                  /*
                   * Check for early end-of-file(EOF) or
                   * end-of-line(EOL)
                   */
                  Else If end-of-file || end-of-line Then
                        Call errorMessage( early EOF or EOL )
                        Return GET_DELETE_EARLY_EOF_OR_EOL_ERROR
                  /*
                   * Add key value to searchKey.  We do not need a
                  final
```

```
         * Else on the options for the code value since
         * previous checks would have eliminated the case
         * before now.
         */
        Else
            If attributeCode == LOWER_POINT_T Then
                Set searchKey.lower.T = value
            Else If attributeCode == LOWER_POINT_X Then
                Set searchKey.lower.X = value
            Else If attributeCode == LOWER_POINT_Y Then
                Set searchKey.lower.Y = value
            Else If attributeCode == LOWER_POINT_Z Then
                Set searchKey.lower.Z = value
            Else If attributeCode == UPPER_POINT_T Then
                Set searchKey.upper.T = value
            Else If attributeCode == UPPER_POINT_X Then
                Set searchKey.upper.X = value
            Else If attributeCode == UPPER_POINT_Y Then
                Set searchKey.upper.Y = value
            Else If attributeCode == UPPER_POINT_Z Then
                Set searchKey.upper.Z = value
            EndIf
        EndIf
    /*
     * Non-key attribute code
     */
    Else
        /*
         * Read non-key attribute value
         */
        Define Char *value;
        Read value
        /*
         * Check for I/O error (low-level fault in stream)
         */
        If I/O error in read Then
            Call errorMessage( error in read )
            Return GET_DELETE_IO_ERROR
        /*
         * Check for early end-of-file(EOF) or
         * end-of-line(EOL)
         */
        Else If end-of-file || end-of-line Then
            Call errorMessage( early EOF or EOL )
            Return GET_DELETE_EARLY_EOF_OR_EOL_ERROR
        /*
         * Create and add new data attribute
         */
        Else
            Define DataAttribute, attribute
            Set attribute = new DataAttribute
            If attribute == NULL Then
                Call errorMessage( memory allocation )
                Return GET_DELETE_ALLOCATION_ERROR
            EndIf
            Set attribute.code  = attributeCode
            Set attribute.value = value
```

```
                    Add attribute to searchNonKey
                EndIf
            EndIf
EndLoop
/*
 * Done
 */
Return GET_DELETE_SUCCESS
```

Test:

A.12.4 getInitCommand

Name:	getInitCommand
Input:	FILE pointer, **file**
Output:	integer fan size, **fan**
Return:	GET_INIT_SUCCESS or GET_INIT_IO_ERROR GET_INIT_EARLY_EOF_OR_EOL_ERROR GET_INIT_INVALID_FAN
Description:	Reads the initialization command from input stream via FILE pointer. Assumes current stream pointer is at beginning of the file and returns file pointer open and current position immediately after command just read. The routine differs from other index command reads in that the command type is read first and the routine is meant as a special case for a first and only read once init command. The current implementation reads only one parameter from the init command which is the fan size. The absence of the fan or an incorrect fan value (< MINIMUM_FAN_SIZE) causes an error to occur.
Calls:	clearLine() errorMessage() System: scanf()

```
Get FILE pointer, file  /* file to read from              */
Get integer, fan        /* fan size read for index        */
Define Char, temp       /* temporary storage for reading  */
/*
 * Read fan
 */
Read fan
/*
 * Check for I/O error (low-level fault in stream)
 */
If I/O error during read Then
            Call errorMessage( error in read )
            Return GET_INIT_IO_ERROR
/*
 * Check for early end-of-file(EOF) or end-of-line(EOL)
 */
Else If end-of-file || end-of-line Then
            Call errorMessage( early EOF or EOL )
            Return GET_INIT_EARLY_EOF_OR_EOL_ERROR
```

```
/*
 * Check for invalid fan value
 */
      Else If fan < MINIMUM_FAN_SIZE Then
            Call errorMessage( Invalid fan size )
            Return GET_INIT_INVALID_FAN
EndIf
/*
 * Clear rest of the line, ignoring any "junk" present
 */
Call clearLine()
/*
 * Done
 */
Return GET_INIT_SUCCESS
```

Test:

A.12.5 getInsertCommand

Name:	getInsertCommand
Input:	FILE pointer, **file**
Output:	data object, **dataObject**
Return:	GET_INSERT_SUCCESS, or GET_INSERT_IO_ERROR GET_INSERT_EARLY_EOF_OR_EOL_ERROR GET_INSERT_UNKNOWN_DATA_OBJECT_TYPE GET_INSERT_ALLOCATION_ERROR
Description:	Reads insert command from input stream via FILE pointer. Assumes current stream pointer is correct, and returns file pointer open and current position immediately after command just read. The file pointer is expected to be at the beginning of the Insert command immediately after the command code. The Insert command consists of the data object type identifier and a complete listing of the attributes (key and non-key) for the object. An error occurs if the object is of an unknown type or any of the attributes are missing. For any error during the read (missing attribute, etc.), the routine will leave the current values of the command attribute list intact, clear the current line, and returns the error code. An Insert command is made of a data object type specifier and a list of the attributes for that type. No attributes can be missing or an error occurs. Each command is carriage return delimited, i.e., one line per command. The command is read by first readings the type, setting a local variable to determine the number of non-key attributes which will be read later in the command line, and then processing the line. The output of the routine is the data object which is ready to be inserted into the index.
Calls:	errorMessage()

System:	scanf()
	strcpy()

```
Get FILE pointer, file         /* file to read insert command */
Get data object, dataObject    /* output data object read in */
Define integer, dataObjectType /* upper bound determined    */
Define integer, upperBound     /* upper bound determined    */
                               /* by data object type       */
/*
 * Read Insert
 *
 * An Insert command is made of a data object type specifier and a
 * list of the attributes for that type.  No attributes can be
 * missing or an error occurs.  Each command is carriage return
 * delimited, i.e., one line per command.  The command is read by
 * first readings the type, setting a local variable to determine
 * the number of non-key attributes which will be read later in
 * the command line, and then processing the line.
 */
Read dataObjectType
/*
 * Check for I/O error (low-level fault in stream)
 */
If I/O error reading Then
            Call errorMessage( error during read )
            Return GET_INSERT_IO_ERROR
/*
 * Check for end-of-file(EOF) or end-of-line(EOL)
 */
Else If end-of-file || end-of-line Then
            Call errorMessage( early EOF or EOL )
            Return GET_INSERT_EARLY_EOF_OR_EOL_ERROR
/*
 * Create data object for entry based on dataObjectType read and
 * set the upperBound to value determined by data object type
 */
Else If dataObjectType == SMALL Then
            Set dataObject = createDataObject( SMALL )
            Set upperBound = NUM_OF_KEY_ATTRIBUTES +
               NUM_OF_SMALL_ATTRIBUTE - 1
Else If dataObjectType == MEDIUM Then
            Set dataObject = createDataObject( MEDIUM )
            Set upperBound = NUM_OF_KEY_ATTRIBUTES +
               NUM_OF_MEDIUM_ATTRIBUTE - 1
Else If dataObjectType == LARGE Then
            Set dataObject = createDataObject( LARGE )
            Set upperBound = NUM_OF_KEY_ATTRIBUTES +
               NUM_OF_LARGE_ATTRIBUTE - 1
/*
 * Unknown data object type
 */
Else
            Call errorMessage( Unknown or invalid object type read )
            Return GET_INSERT_UNKNOWN_DATA_OBJECT_TYPE
EndIf
/*
 * Check for memory allocation error
```

```
 */
If dataObject == NULL Then
            Call errorMessage( allocation error )
            Return GET_INSERT_ALLOCATION_ERROR
EndIf
/*
 * The attributes in the Insert command come in the proper order,
 * i.e., attribute one is first, attribute two is second, etc.
 * Thus, the first NUM_OF_KEY_ATTRIBUTES are read first and added
 * to the command attribute list.
 */
Loop for I = 0 to NUM_OF_KEY_ATTRIBUTES - 1
            /*
             * Read key value
             */
            Define Float value
            Read value
            /*
             * Check for I/O error (low-level fault in stream)
             */
            If I/O error during read Then
                  Call errorMessage( error during read )
                  Return GET_INSERT_IO_ERROR
            /*
             * Check for end-of-file(EOF) or end-of-line(EOL)
             */
            Else If end-of-file || end-of-line Then
                  Call errorMessage( early EOF or EOL )
                  Return GET_INSERT_EARLY_EOF_OR_EOL_ERROR
            /*
             * Place value of key attribute into proper spot
             */
            Else If
                  Assert that value is a Float
                  Set dataObject.attributes[I].key = value
            EndIf
EndLoop
/*
 * The rest of the attributes are non-key and are also in the
 * "proper" order, i.e., numerical order.  The number of
 * attributes is determined by the data object type (read at
 * beginning of line) and is checked against where the actual line
 * ends.  If the end of line is reached before the total number of
 * attributes required are read, an error occurs.  After the
 * correct number of attributes are read, the FILE pointer is set
 * to the end of the current line.  This ignores any values "left
 * over", but is robust by ignoring extra whitespace or other junk
 * which may cause an inappropriate error.
 */
Loop for I = NUM_OF_KEY_ATTRIBUTES to upperBound
            Define Char value /* non-key character sequence */
            Read value
            /*
             * Check for I/O error (low-level fault in stream)
             */
            If error during read Then
                  Call errorMessage( error during )
```

```
            Return GET_INSERT_IO_ERROR
      /*
       * Check for end-of-file(EOF) or end-of-line(EOL)
       */
      Else If end-of-file || end-of-line Then
            Call errorMessage( early EOF or EOL )
            Return GET_INSERT_EARLY_EOF_OR_EOL_ERROR
      /*
       * Add non-key attrirubte to proper position
       */
      Else If

            Set dataObject.attributes[I].nonKey = value
      EndIf
EndLoop
/*
 * Clear rest of the line, ignoring any "junk" present
 */
Clear rest of line
/*
 * Done
 */
Return GET_INSERT_SUCCESS
```

Test:

A.12.6 getNextCommandCode

Name:	getNextCommandCode
Input:	FILE pointer, **file**
Output:	command code, **command**
Return:	GET_NEXT_COMMAND_CODE_SUCCESS or GET_NEXT_COMMAND_CODE_IO_ERROR GET_NEXT_COMMAND_INVALID_COMMAND
Description:	Reads next command code from file and returns the command value. The file input is assumed to be a DIS Benchmark Suite: Data Management data set file where each line of the file is a separate command to read. There are four valid values for the command code which correspond to the database commands (insert, query, and delete) and a NONE type which indicates that the data file is "empty", i.e., all of the commands in the file have been processed. A command code of NONE does not indicate an error condition which would be returned by an error flag.
Calls:	clearLine() errorMessage() System: scanf()

```
Get FILE pointer, file                    /* file to read          */
Get integer, command                      /* command code read     */
Define integer, error                     /* error flag            */
Define CommandType, commandType           /* command type read     */
/*
 * Read from file until a valid command code is obtained, end-of-
 * file, or a I/O error occurred.  A valid command code is one
```

```
 * which matches one of the three types allowed for a database
 * command.  The end-of-file(EOF) condition is used to indicate
 * that the dataset file has been completely read and the
 * application can terminate.  A special code, NONE, is used to
 * inform the end-of-file condition to the calling routine.
 */
Loop while command is not valid
            Read commandType
            /*
             * Check for I/O error (low-level fault in stream)
             */
            If I/O error during read Then
                  Call errorMessage( error during read )
                  Return GET_NEXT_COMMAND_CODE_IO_ERROR
            /*
             * End-of-file condition
             */
            Else If end-of-file Then
                  /*
                   * No commands left in file - termination
                   */
                  Set command = NONE
            /*
             * End-of-line condition
             */
            Else If end-of-line Then
                  /*
                   * Blank line - ignore
                   */
                  Call errorMessage( Blank line )
                  Set command = INVALID
                  Return GET_NEXT_COMMAND_INVALID_COMMAND
            /*
             * Command type read. Switch on type to read an Insert,
             * Query, or Delete command.
             */
            Else If commandType is INSERT Then
                  /*
                   * Set command to type read
                   */
                  Set command = INSERT
            Else If commandType is QUERY Then
                  /*
                   * Set command to type read
                   */
                  Set command = QUERY
            Else If commandType is DELETE Then
                  /*
                   * Set command to type read
                   */
                  Set command = DELETE
            /*
             * Non-fatal condition: unknown command type.  Unknown
             * commands cause a WARNING to be sent to the error handler
             * and are ignored.  The line is cleared from the input,
and
             * another attempt is made to read a valid command.
```

```
            */
        Else
            /*
             * Notify user of condition, clear rest of line, and
             * set command to invalid value to allow loop to
             * continue
             */
            Call errorMessage( Invalid command code )
            Call clearLine()
            Set command = INVALID
            Return GET_NEXT_COMMAND_INVALID_COMMAND
        EndIf
EndLoop
/*
 * Done
 */
Return GET_NEXT_COMMAND_CODE_SUCCESS
```

Test:

A.12.7 getQueryCommand

Name:	getQueryCommand
Input:	FILE pointer, **file**
Output:	index key, **searchKey** list of non-key values as data attributes, **searchNonKey**
Return:	GET_QUERY_SUCCESS or GET_QUERY_IO_ERROR GET_QUERY_INVALID_CODE_ERROR GET_QUERY_EARLY_EOF_OR_EOL_ERROR GET_QUERY_ALLOCATION_ERROR
Description:	Reads query command from input stream via FILE pointer. Assumes current stream pointer is correct, and returns file pointer open and current position immediately after command just read. The file pointer is expected to be at the beginning of the Query command immediately after the command code. The Query command consists of a list of attribute code and value pairs. An error occurs if any attribute code does not have an accompanying attribute value. For any error during read (missing pair, etc.), the routine will leave the current values of the command attribute list intact and clear the FILE pointer to the end of the current line. A Query command is made up of a list of attribute code and value pairs. The total number of pairs can range from zero to the MAX_ATTRIBUTE_CODE. Each command is carriage return delimited, i.e., one line per command. So, to read the Query command, read until the line and store the attribute code/value pairs as we go along. It is possible to have a query command return an empty attribute list without producing an error. Since missing attributes are defaulted to wild-

	card values, this type of query would logically return the entire database. This type of query is a valid query comand as defined by the specification.
Calls:	clearLine() errorMessage() System: scanf() strcpy()

```
Get FILE pointer, file                  /* file to read          */
Get index key, searchkey                /* key part of command   */
Get non-key DataAttribute list, searchNonKey    /* non-key part */
                                                /* of command   */

/*
 * Set searchKey to wild-card values (max or min)
 */
Set searchKey.lower.T    = MINIMUM_VALUE_OF_FLOAT
Set searchKey.lower.X    = MINIMUM_VALUE_OF_FLOAT
Set searchKey.lower.Y    = MINIMUM_VALUE_OF_FLOAT
Set searchKey.lower.Z    = MINIMUM_VALUE_OF_FLOAT

Set searchKey.upper.T    = MAXIMUM_VALUE_OF_FLOAT
Set searchKey.upper.X    = MAXIMUM_VALUE_OF_FLOAT
Set searchKey.upper.Y    = MAXIMUM_VALUE_OF_FLOAT
Set searchKey.upper.Z    = MAXIMUM_VALUE_OF_FLOAT
/*
 * Set non-key attribute list to empty which clears out anything
 * which was left over from previous reads or from initialization.
 */
Set searchNonKey = EMPTY
/*
 * Check assumptions for attribute code values
 */
Assert that NUM_OF_KEY_ATTRIBUTES < total number of attributes
      Assert that every key attribute code < every non-key attribute
         code
/*
 * Read Query command
 *
 * A Query command is made up of a list of attribute code and
 * value pairs.  The total number of pairs can range from zero to
 * the MAX_ATTRIBUTE_CODE.  Each command is carriage return
 * delimited, i.e., one line per command. So, to read the Query
 * command, read until the line and store the attribute code/value
 * pairs as we go along.
 */
While not at end-of-file or end-of-line
            Define integer attributeCode
            Read attributeCode
            /*
             * Check for I/O error (low-level fault in stream)
             */
            If I/O error during read Then
                  Call errorMessage( error in read )
                  Return GET_QUERY_IO_ERROR
            /*
             * Check for end-of-file(EOF) or end-of-line(EOL)
             */
```

```
Else If end-of-file || end-of-line Then
      /*
       * Normal termination of Query read
       */
      Return GET_QUERY_SUCCESS
/*
 * Check for invalid attribute code
 */
Else If attributeCode < MIN_ATTRIBUTE_CODE or attributeCode
   > MAX_ATTRIBUTE_CODE Then
      Call errorMessage( WARNING: Invalid attribute code )
      Call clearLine()
      Return GET_QUERY_INVALID_CODE_ERROR
/*
 * Check code for key attribute value
 */
Else If attributeCode < NUM_OF_KEY_ATTRIBUTES Then
      /*
       * Read key attribute value
       */
      Define Float value
      Read value
      /*
       * Check for I/O error (low-level fault in stream)
       */
      If I/O error in read Then
            Call errorMessage( error in read )
            Return GET_QUERY_IO_ERROR
      /*
       * Check for early end-of-file(EOF) or
       * end-of-line(EOL)
       */
      Else If end-of-file || end-of-line Then
            Call errorMessage( early EOF or EOL )
            Return GET_QUERY_EARLY_EOF_OR_EOL_ERROR
      /*
       * Add key value to searchKey.  We do not need a
      final
       * Else on the options for the code value since
       * previous checks would have eliminated the case
       * before now.
       */
      Else
            If attributeCode == LOWER_POINT_T Then
                  Set searchKey.lower.T = value
            Else If attributeCode == LOWER_POINT_X Then
                  Set searchKey.lower.X = value
            Else If attributeCode == LOWER_POINT_Y Then
                  Set searchKey.lower.Y = value
            Else If attributeCode == LOWER_POINT_Z Then
                  Set searchKey.lower.Z = value
            Else If attributeCode == UPPER_POINT_T Then
                  Set searchKey.upper.T = value
            Else If attributeCode == UPPER_POINT_X Then
                  Set searchKey.upper.X = value
            Else If attributeCode == UPPER_POINT_Y Then
                  Set searchKey.upper.Y = value
```

```
                    Else If attributeCode == UPPER_POINT_Z Then
                          Set searchKey.upper.Z = value
                    EndIf
              EndIf
        /*
         * Non-key attribute code
         */
        Else
              /*
               * Read non-key attribute value
               */
              Define Char *value;
              Read value
              /*
               * Check for I/O error (low-level fault in stream)
               */
              If I/O error in read Then
                    Call errorMessage( error in read )
                    Return GET_QUERY_IO_ERROR
              /*
               * Check for early end-of-file(EOF) or
               * end-of-line(EOL)
               */
              Else If end-of-file || end-of-line Then
                    Call errorMessage( early EOF or EOL )
                    Return GET_QUERY_EARLY_EOF_OR_EOL_ERROR
              /*
               * Create and add new data attribute
               */
              Else
                    Define DataAttribute, attribute
                    Set attribute = new DataAttribute
                    If attribute == NULL Then
                          Call errorMessage( memory allocation )
                          Return GET_DELETE_ALLOCATION_ERROR
                    EndIf
                    Set attribute.code      = attributeCode
                    Set attribute.value     = value
                    Add attribute to searchNonKey
              EndIf
        EndIf
EndLoop
/*
 * Done
 */
Return GET_QUERY_SUCCESS
```

Test:

A.12.8 openFiles

Name:	openFiles
Input:	input file name, **inputFileName** output file name, **outputFileName** metrics file name, **metricsFileName**

Output:	input FILE pointer, **inputFile** output FILE pointer, **outputFile** metrics FILE pointer, **metricsFile**
Return:	OPEN_FILES_SUCCESS or OPEN_FILES_INPUT_FILE_ERROR OPEN_FILES_OUTPUT_FILE_ERRROR OPEN_FILES_METRIC_FILE_ERROR
Description:	Opens files in system using input file name. The FILE pointers are left open and return for reading from the beginning of the files.
Calls:	errorMessage() System: fopen()

```
Get input file name,     inputFileName
Get output file name,    outputFileName
Get metrics file name,   metricsFileName
/*
 * Input File
 */
Set inputFile = fopen( inputFileName, readOnly )
If inputFile is NULL Then
            Call errorMessage( Can't open inputFileName for reading )
            Return OPEN_FILES_INPUT_FILE_ERROR
EndIf
/*
 * Output File
 */
      Set outputFile = fopen( outputFileName, writeOnly )
If outputFile is NULL Then
            Call errorMessage( Can't open outputFileName for writing )
            Return OPEN_FILES_OUTPUT_FILE_ERRROR
EndIf
/*
 * Metric File
 */
      Set metricsFile = fopen( metricsFileName , writeOnly )
If metricsFile is NULL Then
            Call errorMessage( Can't open metricsFileName for writing )
            Return OPEN_FILES_METRIC_FILE_ERROR
EndIf
/*
 * Done
 */
Return OPEN_FILES_SUCCESS
```

Test:

A.12.9 outputMetricsData

Name:	outputMetricsData
Input:	metric file, FILE **metricFile** Metric struct, **metrics**
Output:	integer flag indicating success or error
Return:	OUTPUT_METRIC_SUCCESS, or

	OUTPUT_METRIC_FAILURE
Description:	
Calls:	calcMetricsData()
	System: fprintf()

```
Get metric file name, metricFile    /* file for metric output */
Get Metric struct, metrics          /* metric struct values */
/*
 * Determine metric performance values
 */
Call calcMetricsData( metrics )
/*
 * Output metric info in nice format
 */

/*
 * Done
 */
Return OUTPUT_METRIC_SUCCESS
```

Test:	

A.12.10 outputQuery

Name:	outputQuery
Input:	output buffer, **outputBuffer** list of data objects, **objects**
Output:	integer flag indicating success or error
Return:	OUTPUT_QUERY_SUCCESS, or OUTPUT_QUERY_FAILURE
Description:	Place list of data objects into output buffer. The routine first converts the data object into a character string "equivalent" as define dby the specification. This equivalent is approximately the same as the Insert command: data object type, eight key values (float), and the appropriate number of non-key values (char strings). The routine will check to see if the addition of the equivalent string will exceed the maximum size of the buffer and flushes the buffer if appropriate.
Calls:	flushOutputBuffer() System:

```
Get output buffer, buffer        /* buffer to place objects    */
Get list of data objects, objecs /* objects to place in buffer */
/*
 * Place each object into output buffer
 */
Loop for every object in list of data objects, objects
            /*
             * Create the string version of the data object.  The
             * "string version" is defined by the spec and closely
             * follows the format for an Insert command, i.e., the data
             * object type, eight key values (floats), and the
             * appropriate number of non-key values (char strings).
             * Need the string version for both output and for checking
```

```
         * if buffer needs to be flushed.
         */
        Define character string, string
        Convert object into string
        /*
         * Check if length of object string will exceed maximum
           size
         * of buffer.
         */
        If length of string + current size of buffer >
           MAX_BUFFER_SIZE Then
              /*
               * Flush the buffer to make room
               */
              Call flushOutputBuffer( buffer )
        EndIf
        /*
         * Add object string to buffer
         */
        Add string to buffer
EndLoop
/*
 * Done
 */
Return OUTPUT_METRIC_SUCCESS
```

Test:

A.13 METRICS

A.13.1 calcMetricsData

Name:	calcMetricsData
Input:	Metric structure, **metrics**
Output:	Metric structure, **metrics**
Return:	void
Description:	Calculate the average and variance of provided metric command structure and store. The routine will only change the **avg** and **deviation** variables. The routine checks that at least one sample time difference is present in the command metric structure which is placed there via the *updateMetricsData* routine. Two errors are possible for this routine where the first error occurs if the routine is called for a structure that does not have at least one sample time difference present. The second error occurs in the event of round-off which may cause the calculated variance to be negative, although this possibility is analytically impossible. In the case of either error, the **avg** and **deviation** members of the command metric structure are set to zero which prevents their later use if "junk" were left there either as initialization or from previous calculations.
Calls:	errorMessage() getTime() System: sqrt()

```
Get Metric, metrics
Define Float, temp
/*
 * total time for application to execute
 */
Set metrics.totalTime = getTime() - metrics.totalTime

/*
 * Calculate metrics for Insert command
 */

/*
 * Check for no samples in command metric
 */
If metrics.insertCommandMetric.numOfCommands == ZERO Then
            Set metrics.insertCommandMetric.avg       = MIN_TIME_VALUE
            Set metrics.insertCommandMetric.deviation = MIN_TIME_VALUE
EndIf
/*
 * Find Average
 */
      Set metrics.insertCommandMetric.avg =
         metrics.insertCommandMetric.sum /
         metrics.insertCommandMetric.numOfCommands
/*
 * Find Standard deviation
 */
      Set temp = metrics.insertCommandMetric.sumSquares -
         metrics.insertCommandMetric.sum *
         metrics.insertCommandMetric.sum /
         metrics.insertCommandMetric.numOfCommands
If temp < ZERO Then
            /*
             * Possible round-off may cause negative variance
             */
            Set metrics.insertCommandMetric.avg       = MIN_TIME_VALUE
            Set metrics.insertCommandMetric.deviation = MIN_TIME_VALUE
Else
            Set metrics.deviation = sqrt( temp / metrics.numOfCommands
               )
EndIf

/*
 * Calculate metrics for Query command
 */

/*
 * Check for no samples in command metric
 */
If metrics.queryCommandMetric.numOfCommands == ZERO Then
            Set metrics.queryCommandMetric.avg        = MIN_TIME_VALUE
            Set metrics.queryCommandMetric.deviation  = MIN_TIME_VALUE
EndIf
/*
 * Find Average
 */
```

```
        Set metrics.queryCommandMetric.avg =
           metrics.queryCommandMetric.sum /
           metrics.queryCommandMetric.numOfCommands
/*
 * Find Standard deviation
 */
        Set temp = metrics.queryCommandMetric.sumSquares -
           metrics.queryCommandMetric.sum *
           metrics.queryCommandMetric.sum /
           metrics.queryCommandMetric.numOfCommands
If temp < ZERO Then
            /*
             * Possible round-off may cause negative variance
             */
            Set metrics.queryCommandMetric.avg        = MIN_TIME_VALUE
            Set metrics.queryCommandMetric.deviation  = MIN_TIME_VALUE
Else
            Set metrics.queryCommandMetric.deviation = sqrt( temp / (
               metrics.queryCommandMetric.numOfCommands - 1 ) )
EndIf

/*
 * Calculate metrics for Delete command
 */

/*
 * Check for no samples in command metric
 */
If metrics.deleteCommandMetric.numOfCommands == ZERO Then
            Set metrics.deleteCommandMetric.avg       = MIN_TIME_VALUE
            Set metrics.deleteCommandMetric.deviation = MIN_TIME_VALUE
EndIf
/*
 * Find Average
 */
        Set metrics.deleteCommandMetric.avg =
           metrics.deleteCommandMetric.sum /
           metrics.deleteCommandMetric.numOfCommands
/*
 * Find Standard deviation
 */
        Set temp = metrics.deleteCommandMetric.sumSquares -
           metrics.deleteCommandMetric.sum *
           metrics.deleteCommandMetric.sum /
           metrics.deleteCommandMetric.numOfCommands
If temp < ZERO Then
            /*
             * Possible round-off may cause negative variance
             */
            Set metrics.deleteCommandMetric.avg       = MIN_TIME_VALUE
            Set metrics.deleteCommandMetric.deviation = MIN_TIME_VALUE
Else
            Set metrics.deleteCommandMetric.deviation = sqrt( temp / (
               metrics.deleteCommandMetric.numOfCommands - 1 ) )
EndIf
/*
 * Done
```

```
 */
Return
```

A.13.2 initMetricsData

Name:	initMetricsData
Input:	Metric structure, **metrics**
Output:	Metric structure, **metrics**
Return:	INIT_METRIC_SUCCESS, or INIT_METRIC_TIMING_ERROR
Description:	This routine initiazes the Metrics module by setting the appropriate values and/or flags for later metric collection. The timing information for total execution time, input time, and output time are set to the current system time returned as timing marks for later processing. Note that the time() system call might fail so a check is provided. This isn't a fatal error in the sense that neither the Database or Input & Output modules will fail, but all metric information collected will be invalid. The time routine is checked only in the *initMetricsData* routine and if all calls are successfull, the rest of the application will assume success. This prevents unnecessary checks and reduces the complexity caused by excess return checks for system calls. The individual command metric data are also initialized to either logical values (number of commands, sum of command times, sum of squares of command times) or to "highly unlikely" values (worst, best, avg, deviation) which should be easily noticed if output. The routine *updateMetricsData* uses the value of the lastCommand field to determine which command metric to update. The first time through, there is no command so an initial value of NONE is placed in the field. The *updateMetricsData* should interpret this value so as not to update an command metrics for that loop.
Calls:	errorMessage() getTime() System:

```
Get Metric, metrics
Define Time, temp
/*
 * Setup timing marks for metric collection.  If the system timing
 * call fails, return an error to the calling routine.  This is
 * not necessarily fatal, but it's difficult to see how later
 * metric info can be taken accurately if this call fails.  This
 * error should not affect the database or input/output routines.
 */
Call temp = getTime()
If error occured taking the system time Then
            /*
             * Inform user of error and exit
```

```
            */
            Call errorMessage( Unable to determine system timing info )
            Return INIT_METRIC_TIMING_ERROR
EndIf
Set metrics.totalTime    = temp
Set metrics.inputTime    = temp
Set metrics.outputTime   = temp
/*
 * Set initial values for each command metric for insert, query,
 * and delete.  The statistical values for the worst, best,
 * average, and standard deviation values are set to "highly
 * unlikely" values which should be easily noticed if unchanged
 * and output as a metric result.
 */
Set metrics.insertCommandMetric.numOfCommands    = 0
Set metrics.insertCommandMetric.sum              = 0.0
Set metrics.insertCommandMetric.sumSquares       = 0.0
Set metrics.insertCommandMetric.worst            = MIN_TIME_VALUE
Set metrics.insertCommandMetric.best             = MAX_TIME_VALUE
Set metrics.insertCommandMetric.avg              = MIN_TIME_VALUE
Set metrics.insertCommandMetric.deviation        = MIN_TIME_VALUE

Set metrics.queryCommandMetric.numOfCommands     = 0
Set metrics.queryCommandMetric.sum               = 0.0
Set metrics.queryCommandMetric.sumSquares        = 0.0
Set metrics.queryCommandMetric.worst             = MIN_TIME_VALUE
Set metrics.queryCommandMetric.best              = MAX_TIME_VALUE
Set metrics.queryCommandMetric.avg               = MIN_TIME_VALUE
Set metrics.queryCommandMetric.deviation         = MIN_TIME_VALUE

Set metrics.deleteCommandMetric.numOfCommands    = 0
Set metrics.deleteCommandMetric.sum              = 0.0
Set metrics.deleteCommandMetric.sumSquares       = 0.0
Set metrics.deleteCommandMetric.worst            = MIN_TIME_VALUE
Set metrics.deleteCommandMetric.best             = MAX_TIME_VALUE
Set metrics.deleteCommandMetric.avg              = MIN_TIME_VALUE
Set metrics.deleteCommandMetric.deviation        = MIN_TIME_VALUE
/*
 * The routine updateMetricsData expects the lastCommand field of
 * the Metric structure to indicate which command metric should be
 * updated.  There is no update the first update, so set field to
 * none which will be interpreted by the updateMetricsData routine
 * as a "bye".
 */
Set metrics.lastCommand = NONE
/*
 * Done
 */
Return INIT_METRIC_SUCCESS
```

A.13.3 setMetricsData

Name:	setMetricsData
Input:	Metric structure, **metrics**
	enum CommandType, **type**

Output:	metric structure, **metrics**
Return:	void
Description:	This routine sets two values for later metric collection. The first is specific to the individual command being processed and sets the lastTimeMark field for that command metric structure. The second is the command type for later update which is stored in the lastCommand field of the **metrics** structure. The routine does not require a return code since no fatal error can occur. If a command type is passed which is not recognized, it informs the user via th errorMessage routine.
Calls:	errorMessage() getTime() System:

```
Get metric struct metrics
Get CommandType, type
/*
 * Set the lastTimeMark and lastCommand values
 */
If type == INSERT Then
            Set metrics.insertCommandMetric.lastTimeMark    = getTime()
            Set metrics.lastCommand = INSERT
Else If type == QUERY Then
            Set metrics.queryCommandMetric.lastTimeMark     = getTime()
            Set metrics.lastCommand = QUERY
Else If type == DELETE Then
            Set metrics.deleteCommandMetric.lastTimeMark    = getTime()
            Set metrics.lastCommand = DELETE
Else
            /*
             * Unknown command type. Non-fatal error which simply
            causes
             * no timing info to be collected for this command
             * processing.
             */
            Call errorMessage( Unknown command type to set )
EndIf

/*
 * Done
 */
Return
```

Test:

A.13.4 updateMetricsData

Name:	updateMetricsData
Input:	Metric struct, **metrics** enum CommandType, **type**
Output:	Metric struct, **metrics**
Return:	void

Description:	Determine time for command to complete via stored time in structure and current system time. The specific command to update is determined by the input command **type**. Note that the specific metric command structure must have the lastTimeMark element correctly stored. Update internal variables of **sum**, **sumSquares**, and **numOfCommands** of the specified command metric structure for later metric calculations. The routine determines the time to complete command by calling the system for the current wall clock time and subtracting the time mark set in the metric structure. If a negative time difference is found, the value is ignored and the metric command structure is not updated. The update consists of incrementing the total number of samples for this command type, the sum of the time differences up to this sample, the sum of the squares of the time differences up to this sample, and the best (fastest) and worst (slowest) time difference. Note that care must be taken for the implementation to insure that overflow conditions do not arise for the sum of the time differences and the sum of the square of the time differences. The implementation of the timing mechanism for the specific Time type will determine the steps necessary to prevent this error condition. The commands Insert, Query, and Delete are the only commands which are updated. A command value of NONE, INVALID, or an unknown command type is ignored. This prevents the corruption of valid data stored for the Insert, Query, or Delete metrics by updating metric data for "failed" commands, i.e., commands which did not function properly but do not cause a fatal error which halts execution, or commands which can't be read from input correctly, etc.
Calls:	errorMessage() getTime() System:

```
Get Metric struct, metrics
Get enum CommandType, type
Define Time, commandTime
/*
 * Find current system time
 */
Call commandTime = getTime()
/*
 * Switch on command type to update
 */
If type == INSERT Then
            /*
             * Determine time of command processing
             */
            Set commandTime = commandTime -
               metrics.insertCommandMetric.lastTimeMark
            If commandTime < ZERO Then
                  Call errorMessage( The timing mark for command does
                     not appear to be properly set - the last time mark
                     appears to be more recent then the current time )
            Else
```

```
                /*
                 * Update stored best and worst values
                 */
                If commandTime < metrics.insertCommandMetric.best
                Then
                      Set metrics.insertCommandMetric.best =
                         commandTime
                EndIf
                If commandTime > metrics.insertCommandMetric.worst
                Then
                      Set metrics.insertCommandMetric.worst =
                         commandTime
                EndIf
                /*
                 * Update sums: Caution should be taken to prevent
                 * overflow conditions.
                 */
                Set metrics.insertCommandMetric.sum =
                   metrics.insertCommandMetric.sum + commandTime
                Set metrics.insertCommandMetric.sumSquares =
                   metrics.insertCommandMetric.sumSquares +
                   commandTime * commandTime
                Set metrics.insertCommandMetric.numOfCommands =
                   metrics.insertCommandMetric.numOfCommands + 1
          EndIf
Else If type == QUERY Then
          /*
           * Determine time of command processing
           */
          Set commandTime = commandTime -
             metrics.queryCommandMetric.lastTimeMark
          If commandTime < ZERO Then
                Call errorMessage( The timing mark for command does
                   not appear to be properly set )
          Else
                /*
                 * Update stored best and worst values
                 */
                If commandTime < metrics.queryCommandMetric.best Then
                      Set metrics.queryCommandMetric.best =
                         commandTime
                EndIf
                If commandTime > metrics.queryCommandMetric.worst
                Then
                      Set metrics.queryCommandMetric.worst =
                         commandTime
                EndIf
                /*
                 * Update sums: Caution should be taken to prevent
                 * overflow conditions.
                 */
                Set metrics.queryCommandMetric.sum =
                   metrics.queryCommandMetric.sum + commandTime
                Set metrics.queryCommandMetric.sumSquares =
                   metrics.queryCommandMetric.sumSquares +
                   commandTime * commandTime
```

```
                Set metrics.queryCommandMetric.numOfCommands =
                   metrics.queryCommandMetric.numOfCommands + 1
            EndIf
Else If type == DELETE Then
            /*
             * Determine time of command processing
             */
            Set commandTime = commandTime -
               metrics.deleteCommandMetric.lastTimeMark
            If commandTime < ZERO Then
                Call errorMessage( The timing mark for command does
                   not appear to be properly set )
            Else
                /*
                 * Update stored best and worst values
                 */
                If commandTime < metrics.deleteCommandMetric.best
                Then
                      Set metrics.deleteCommandMetric.best =
                         commandTime
                EndIf
                If commandTime > metrics.deleteCommandMetric.worst
                Then
                      Set metrics.deleteCommandMetric.worst =
                         commandTime
                EndIf
                /*
                 * Update sums: Caution should be taken to prevent
                 * overflow conditions.
                 */
                Set metrics.deleteCommandMetric.sum =
                   metrics.deleteCommandMetric.sum + commandTime
                Set metrics.deleteCommandMetric.sumSquares =
                   metrics.deleteCommandMetric.sumSquares +
                   commandTime * commandTime
                Set metrics.deleteCommandMetric.numOfCommands =
                   metrics.deleteCommandMetric.numOfCommands + 1
            EndIf
End
```

Appendix B: DIS Benchmark Suite: Image Understanding Software Design

B. DESIGN DESCRIPTION

This document describes the baseline software design of the Image Understanding benchmark for the DIS Benchmark Suite. The document is separated into two parts where the first part, Design Description, presents the design approach at a high-level, describes various implementation decisions, and explains the interaction between the individual function descriptions. The second part, Pseudo-code, presents the low-level descriptions of the individual routines and methods which make up the baseline application.

The initial input to and the final output from the Image Understanding benchmark is fixed as specified in [AAEC-1]. Any intermediate results or design choices are picked as an example of only one of many possible. Anyone implementing the benchmark is free to design the internals differently and is even encouraged to, especially to gain accuracy or speed. The implementation can be modified, but the functionality of the benchmark must remain intact and will be tested by validating the final output.

B.1 GOALS

Several goals were established and followed for this design:

- Accurately follow the Image Understanding specification as described in the DIS Benchmark Suite [AAEC-1].

- Strike a balance between easily understandable source code and optimized source code. Optimization usually has the effect of making software implementation more complex or more difficult to immediately understand. However, the baseline design should represent a "reasonable" image understanding application and the baseline performance figures should be comparable to a real application which indicates some optimization. The choice of whether to use a specific optimization technique is determined by the amount of performance increase versus the degradation of the general user to determine the underlying process.

- Provide a complete application which is reasonably robust for a general execution. The application should handle generic errors (unable to locate input files, memory allocation failures, etc.) gracefully, and return control to calling process without an abrupt failure. The application should not be expected to handle hardware specific or special errors unique to a platform or hardware configuration. Also, when errors would require a large amount of code to detect and/or fix, the design reverts to the primary goal of optimization/understandability.

- Provide baseline source code which is relatively easy to understand and modify to a particular implementation approach. The design should allow the alteration of the underlying image understanding algorithms without causing a major shift in the design paradigm.

- Follow the DIS Benchmark C Style Guide [AAEC-2] for the development phase of the baseline source code. The style guide lists several aspects of the source code which can be incorporated into the design segment.

B.2 CONVENTIONS

Several conventions are used in this document to clarify the design and implementation of the benchmark.

- A function name will be *italicized* when referenced within a text setting.
- Structures are in **bold** type when referenced within a text setting.
- Control flow in diagrams is denoted by arrows on solid lines.
- Subroutines are denoted by "bold" rectangular boxes.

B.3 OVERVIEW

The Image Understanding Sequence defined in [AAEC-1], is duplicated in Figure B-1, and consists of the following: a morphological filter component, a region of interest (ROI) selection component, and a feature extraction component. The morphological filter component provides a spatial filter to remove background clutter in the image. Next, the ROI selection component performs a thresholding to determine target pixels, groups these pixels into ROIs, and selects a subset of ROIs based on specific selection logic. Finally, the feature extraction component operates over and computes features for these selected ROIs.

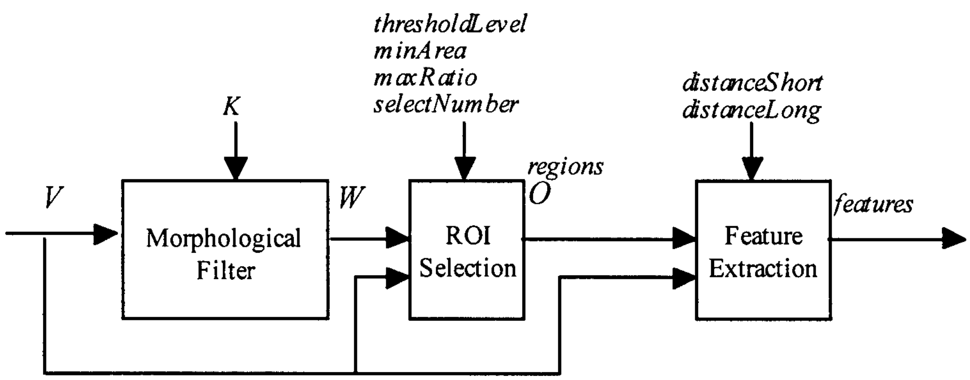


Figure B-1: Image Understanding Sequence

The input required by the sequence is a set of parameters, an image, *V*, and a kernel, *K*. The first step in the sequence is a spatial morphological filter component generating image *W*. Then, the ROI selection component performs a thresholding and groups connected pixels into ROIs (or targets) contained in image *W*. This component then computes initial features for each ROI in image *W*, and selects a subset of ROIs based on the values of these features. These selected ROIs are stored in object image, *O*, and the initial features for each selected ROI are included in list, *regions*. Lastly, the feature extraction component computes additional features for the selected ROIs. The output at the end of the sequence is a feature list, *features*, with both sets of features computed for each selected ROI. Additional information and details regarding the sequence are in [AAEC-1].

The output O does not need to be an image, but does need to contain enough information so that each selected ROI is differentiated from other ROIs, and so each pixel within an ROI can be referenced. In the baseline implementation of the benchmark, this is achieved by having O be an image using the same memory as the intermediate filtered image W. The depth of the values in O is driven by the maximum number of ROIs possible ($(2^{16} - 2)$ as discussed in [AAEC-1]). Thus this implementation reuses the memory for the filtered image W to store the labeled image O.

Others implementing the benchmark are free to design O to be something other than an image as long as the utility of O does not change. For example, instead of a complete image containing all the ROIs labeled with a distinct index, a subimage or chip could be extracted for each ROI where the chip boundaries could be the smallest rectangular region that would contain that ROI. Then a method of obtaining the location of the ROI relative the filtered image W must also be retained (i.e., an offset to place the chip over the proper location in W). In this manner, there would be *selectNumber* chips and offsets to specify the selected ROIs providing a convenient mechanism to parallelize the process. As another example, an ROI may be specified as a list of pixel locations where an ROI with twenty-five pixels would have a list of twenty-five pixel locations relative to the pixel coordinates of W. Then, output O would not be an image at all, but would be a list of ROIs where each element on the list is also a list (of pixel values for that particular ROI). The implementation chosen for the baseline, where O is a labeled image, was chosen to simplify the presentation.

A functional hierarchy for the baseline design is provided in Figure B-2. Each function is represented by a box and is connected in the hierarchy by a dotted arrow starting at the calling function and ending at the function called. Input parameters and intermediate data that is named in Figure B-1 is also included with a solid arrow from the function that creates or reads in the data to the function(s) that use the data. The *transposeFlag* parameter, discussed below (but not included in Figure B-1), is displayed in the functional hierarchy with a dashed arrow. This parameter is set within the *readInput* function and used within the *writeOutput* function.

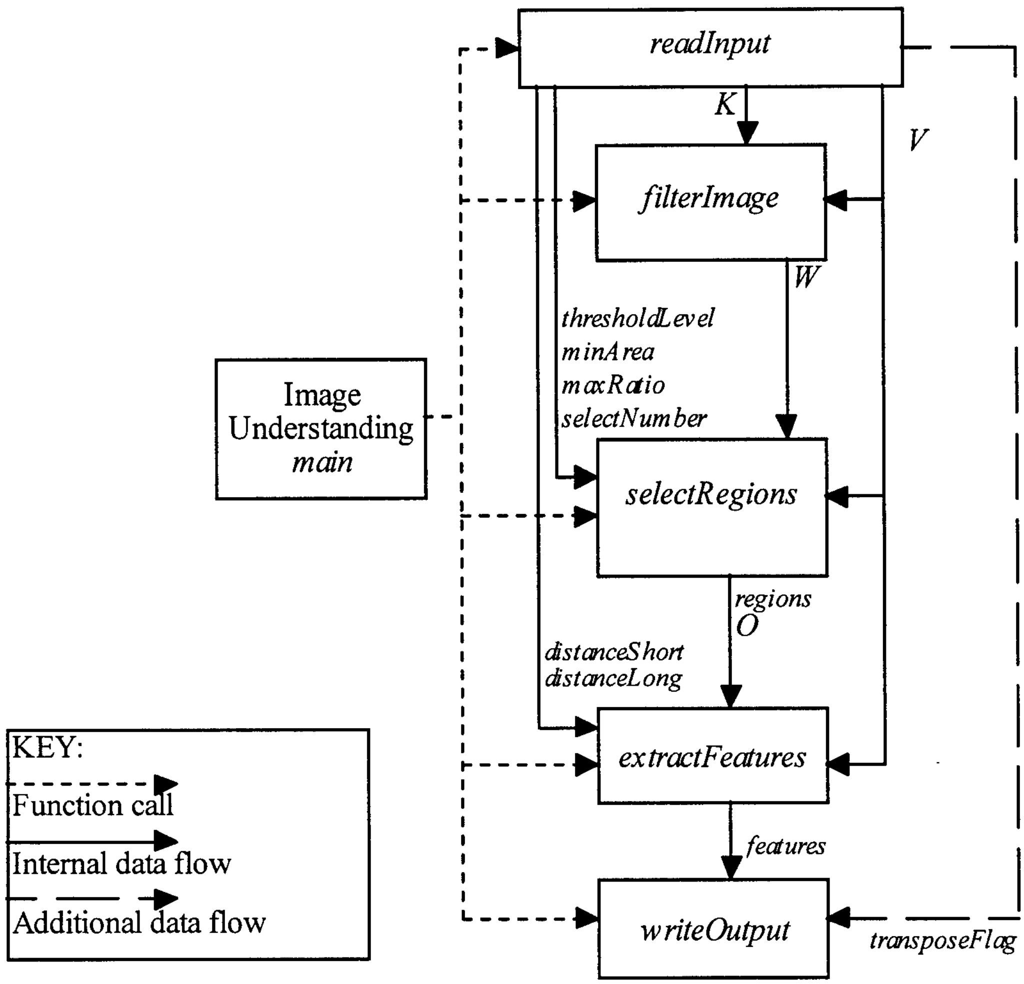


Figure B-2: Function Overview

This baseline implementation of the sequence has been designed favoring unit stride along a row (or imagery stored in row dominant order). Since the filtering component of the sequence uses a kernel to define an area around the pixel being processed, more than one row of the image is accessed to process a single pixel. Furthermore, the filtering component of the sequence is implemented to perform calculations in-place which requires some temporary storage. For an image with X columns by Y rows and a kernel with M columns and N rows, this temporary buffer size must be at least X*N/2 pixel-sized memory units. Consider the case when the image is a long thin rectangular strip where X is much larger than Y. Then, the amount of memory required for the temporary buffer, X*N/2, can be significant. What if it were able to lower that amount to Y*N/2? Then the savings in the amount of cache required would be great. If it were possible to transpose the image before processing it, this saving could be realized.

Analysis of the sequence shows that transposing the image will not affect the output from the sequence as long as: 1) the kernel is also transposed before the filtering component, and 2) the features dependent on the orientation of the coordinate system – *centroid, GLCM entropy*, and *GLCM energy* – are changed so that they reflect values that would be obtained if the transpose had not occurred. Fortunately, "transposing" these features after the fact is simple. The centroid row and column values need to be switched and the directions for the GLCM descriptors are not affected or can be paired so that a simple switch will work (see Section B.4.1 for more details). Since it is possible to generate accurate output from the process executed on a transposed image, the baseline implementation invokes an image transpose to minimize the temporary memory requirements. This method has an additional advantage of minimizing the cache required. Therefore, upon input, if the number of columns is larger than the number of rows, then the image is transposed. A *transposeFlag* parameter will be set to reflect that the image has been transposed. If the transpose has occurred, then the kernel is also transposed. In addition, the output function, *writeOutput*, must "transpose" the output to obtain values as if the transpose had not been done.

The only function which does not have a calling function in the functional hierarchy shown in Figure B-2, is the *main* function whose calling function is the user or system. Note that, although an attempt was made to prevent lines in the diagrams from overlapping, this was not avoidable in all circumstances. However, the correct hierarchy can be determined with logical reasoning. This hierarchy represents a top level flow, where the major functions are included at this level. Four major modules have been designed to implement this benchmark: `Input & Output`, `Filtering`, `Region Selection`, and `Feature Extraction`. The `Input & Output` module, which controls the flow of the program including input and output, contains the *main* program along with the routines *readInput* and *writeOutput*. The `Filtering` module implements the Morphological Filtering component of the sequence with function *filterImage*. The `Region Selection` module implements the ROI Selection component of the sequence using function *selectRegions*. The final `Feature Extraction` module, implements the Feature Extraction component of the sequence with function *extractFeatures*. Details of the functions used for each module, as well as any supporting functions, are included in the sections below.

All input and output is isolated to the `Input & Output` module to allow ease in installing the sequence onto multiple hardware platforms. The metrics for the sequence, which provide timing information for the baseline performance, are also included in this module.

One of the goals for the Image Understanding software design is to produce a robust application which will gracefully handle most errors. The strategy by which these errors are handled is applied uniformly in the application. The primary approach consists of the return of integer codes from each routine which can fail. A routine will have a successful return code, indicating that the specific task required of the routine was accomplished, or one or more error return codes which indicate the specific error that occurred. The return code indicates the state of the process and any other output data and not necessarily that no error occurred during the execution of the routine. For example, if an error occurs during a subroutine but the subroutine recovers, a successful code is returned to the calling process indicating that all output data and the current execution thread can proceed normally. This approach implies several characteristics for each routine:

- Each routine will handle all "local" errors. A local error is one caused directly by the system, e.g., opening files, reading/writing to/from a stream, etc., or by calls to other functional subroutines. This does not indicate that a routine will abort the execution thread, rather the routine will return an appropriate error code to the calling function.

- Each routine will "clean-up" before the return. If an error occurred, any memory allocated during the execution of the routine will be unallocated.

The only exceptions to the return code approach for the baseline design is for system routines which prescribe a different method and for baseline routines which closely mimic these system calls. The best examples are the memory allocation functions which return a pointer to the allocated memory or NULL if an error occurred. Each routine which can fail, lists the success and error return codes as part of the function description. A complete listing of the return codes for the baseline application is given in the Section B.6.

All error and/or unusual conditions which arise during execution of the application will have a descriptive comment placed in the error stream preceded by the names of the routines where the condition occurred, i.e., an error occurring within subroutine *B* which was called by subroutine *A* would have the message,

A> B> error message

where each subroutine name and the actual message are separated by ">". The message system is accomplished by two routines: *errorMessage* and *flushErrorMessage*. A local buffer is kept by the routines to allow storage of the message and routine names before flushing and the size of the buffer is set such that exceeding the limit is extremely rare. The extreme case when the prepended message is larger than the buffer size will cause the *errorMessage* routine to immediately flush the current error buffer along with a message indicating the premature flush. The *errorMessage* routine inputs two parameters: (1) A character string which should either contain a text representation of the condition that occurred or the name of a routine which should be prepended to the current message, (2) An boolean value indicating the the first parameter message should replace the current error buffer contents, or the first parameter message should be prepended to the current error buffer contents. The *flushErrorMessage* routine takes no parameters as input and simply places the current contents of the error buffer into the standard error stream. A call to the *flushErrorMessage* does not clear the buffer contents.

B.4 IMAGE UNDERSTANDING

The image understanding sequence is composed of three components: a morphological filter component (`Filtering`), an ROI selection component (`Region Selection`), and a feature extraction component (`Feature Extraction`). Modules that implement these components are referenced in parenthesis following the corresponding component name. The algorithms and descriptions for the components are presented in the DIS Benchmark Suite[AAEC-1] document and are not repeated here. In the design of this baseline, it is assumed that the input images will be very large in size and that the kernels will be small relative to the image size. An additional module, `Input & Output`, consists of: the mainline *(main)*, the input and output routines, *readInput* and *writeOutput,* the error handling routines, and the routines calculating metrics. Any interface required by the sequence with peripheral devices is localized within this module.

These four modules are grouped for convenience and are not necessarily isolated from each other. There are data structures and functions that are shared among more than one module. This overlap for each module is discussed in the sections below.

B.4.1 Input & Output Module

This module controls all input and output for the sequence and contains the mainline, *main*, as well as supporting functions. Input to this module is a single input file containing all inputs required by the sequence. Output from this module consists of three files, an output file, a metric file, and an error file. The output file contains an ASCII table of feature values for each selected ROI, the metric file contains metrics for the evaluating performance of the implementation, and the error file contains ASCII messages pertaining to any warnings and/or errors that occurred during execution of the sequence. Three of these files (input, output, and metric) can be specified at runtime to be disk files or can default to be the streams standard input, standard output, and standard error. The error file will always be defined to go to standard error. The format for the input and output file is specified in the DIS Benchmark Suite document [AAEC-1].

A function hierarchy is depicted below in Figure B-3. All the functions contained within the `Input & Output` module are included. Functions contained in other modules, but called within this module are also included and are shaded to signify that they are functions called outside of the module. Both dotted and solid arrows are used starting at the calling function and ending at the function called. Two types of arrows were used only to make the path of overlapping arrows clear. There is no difference between a solid or a dotted arrow. The calls to the other modules highest level functions (*filterImage, selectRegions,* and *extractFeatures*) were left at the highest level only. The lower level functions for these modules are discussed in the sections covering those particular modules.

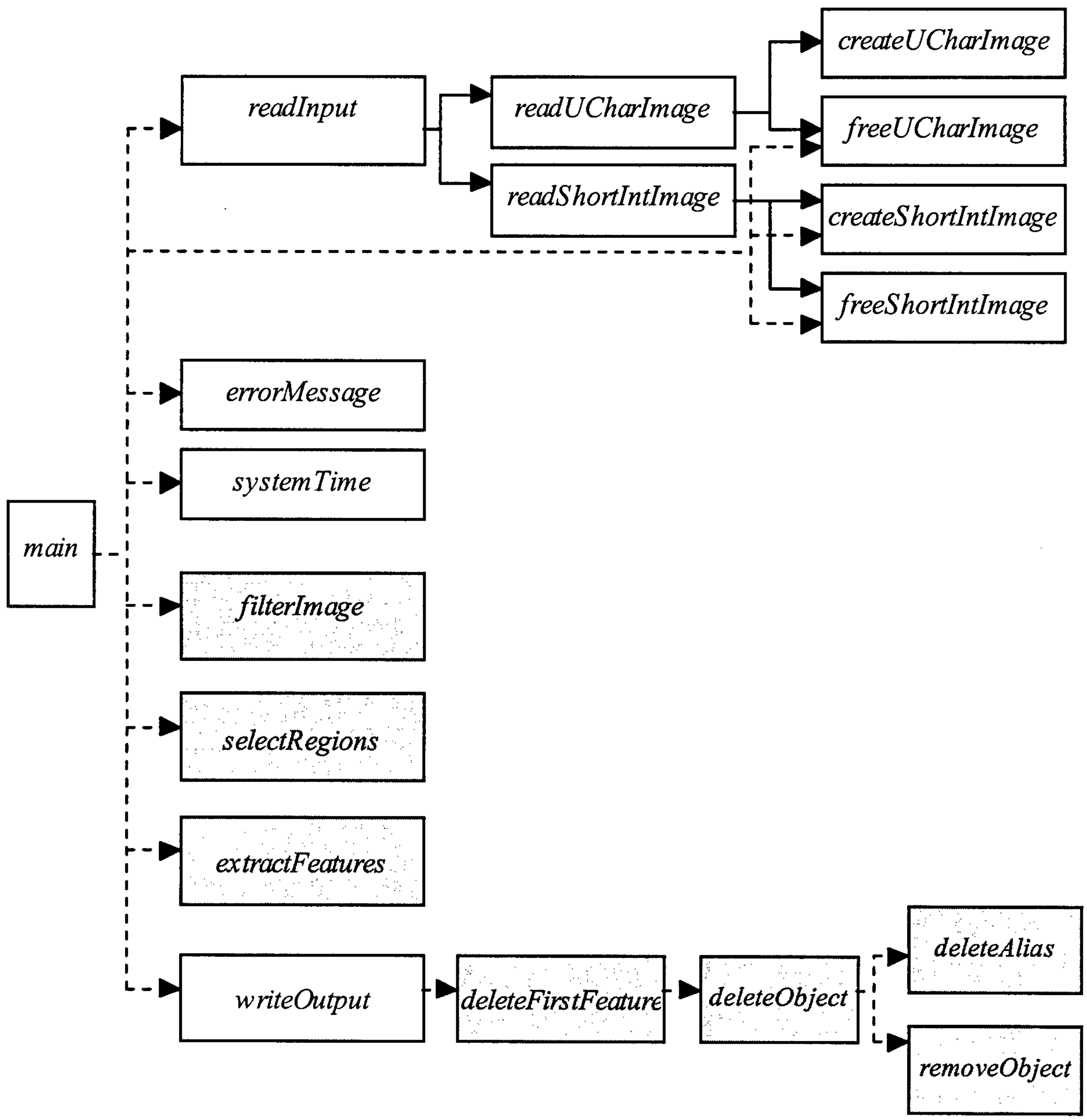


Figure B-3: Input & Output Module Function Hierarchy

All input to the sequence is handled within the routine *readInput*. This includes deciding whether to transpose the input image as well as checking the ranges of the input parameters. When the dimensions of the input image V are read in, if the number of columns is larger than the number of rows, then the image data is stored transposed as it is being read in from input. Thus, although the image is provided in row dominant order, the image is stored in column dominant order. Transposing while reading in the image V is more efficient (for memory access and usage) than reading in the image in row dominant order and then transposing it into column dominant order. The internal parameter, *transposeFlag*, is set to TRUE if this transpose occurs so that the kernel, K, is also transposed upon input. If the number of columns of image V is not larger than the number of rows, the image V and the kernel K are read in and stored in row dominant order, as provided, and the *transposeFlag* is set to FALSE.

The input parameters are checked for validity within *readInput* to catch invalid values as early as possible. The validation tests for the input kernel K are: 1) the kernel is smaller than the input image V (in each dimension), 2) the kernel has dimensions that are odd, and 3) kernel has at least one non-zero pixel value (defining the shape of the kernel).

The other input parameters are checked to ensure that: 1) *minArea* is greater than or equal to zero, 2) *maxRatio* is greater than zero, 3) *selectNumber, distanceShort,* and *distanceLong* are positive non-zero values, 4) *minArea* < total area of input image V, 5) *distanceShort* and *distanceLong* are both less than each dimension of input image V, and 6)*selectNumber* < total number of possible objects (limited to $(2^{16} - 2)$, see Section B.3).

Once the input routine *readInput* accepts the parameters, these values are not checked and are assumed to be valid in lower level routines. If there is an invalid input in the input file, *readInput* returns an error code that is interpreted by the main program and the sequence execution halts.

The output for the sequence consists of writing a table of the features calculated for each selected ROI in the *features* list and is handled in the routine *writeOutput.* If the *transposeFlag* is set to TRUE, then the feature values are "transposed" to reflect features that are valid for input images that are NOT transposed. The flag is set when the input image is transposed before applying the image understanding sequence. Therefore the output must be changed to obtain the values that correspond to applying the sequence to an input image not transposed. This entails a simple switching of values. The features calculated are symmetric with respect to a transpose - the centroid column value becomes the centroid row value (and the centroid row value becomes the centroid column value). When the centroid column and row values are switched, they represent the centroid that would have been derived from the image if the image had NOT been transposed. The GLCM descriptor features have a similar pairing because of the relationship between the directions chosen. For these features, if the transpose has occurred, the 0 degree and the 90 degree descriptor values are switched. The GLCM descriptor values for 45 degrees and 135 degrees are unaffected by the transpose. A 45 degree vector is along the axis of the transpose and remains 45 degrees. A 135 degree vector transposed becomes a 315 degree vector, which is along the same ray as 135, but in the opposite direction. Since the GLCM descriptors are being calculated using sum and difference histograms that are equivalent for directions 180 degrees apart, no change is required for the 135 degree descriptors. Once these swaps are performed, the full set of features represents values for an image that is not transposed. The other features – *area, perimeter, mean,* and *variance* – are not affected by the transposition of the input image. Then, if the *transposeFlag* is set to TRUE, the features are "transposed" (or switched with their matching pair) before output is written. If the *transposeFlag* is set to FALSE, the features that are derived are written out. Either way, the features written to the output file represent features derived on the input image that has not been transposed.

B.4.2 Filtering Module

The `Filtering` module implements the morphological filter component of the sequence and can be accessed through routine *filterImage* as shown in Figure B-2. The inputs to this module are an input image V and a kernel K. From these inputs, the filter is applied which generates a filtered output image, W. The mathematical definition for this filter is given in the specification and repeated here for convenience. Define the morphological operations, *erosion* ($\ominus$) and *dilation* ($\oplus$) as follows:

$$[V \ominus K] = MIN[v(x+m,\ y+n)] \qquad m,n \in Ros(K),\ k(m,n) \neq 0 \qquad (B.4.2.1a)$$

$$[V \oplus K] = MAX[v(x+m,\ y+n)] \qquad m,n \in Ros(K),\ k(m,n) \neq 0 \qquad (B.4.2.1b)$$

where each output pixel is computed at location (x, y) for a morphological kernel, K, which has a local region of support (Ros) that defines its geometric filtering properties with M columns and N rows. For these primitive morphological operations, MAX and MIN are computed locally for every pixel. Only nearby pixels are required to compute output pixels, specifically for the pixels in K that are non-zero.

For this benchmark, the morphological filter is defined as follows. As shown in Figure B-1, V is the input image and W is the output, where

$$W = V - [(V \ominus K) \oplus K] \qquad (B.4.2.2)$$

Since the input image V is used as input for components following this one, it is considered to be provided in a READONLY buffer. This module must be given, in addition to the input image V, and the kernel K, an output buffer for image W. The implementation of the filter defined in Equation *(B.4.2.2)* contains two steps and is designed to require minimal extra memory than what is provided by the calling function.

A function hierarchy for the `Filtering` module is shown in Figure B-4 below. The filtering process is broken down into two steps. The erosion is performed in function *erodeImage*, and the dilation is performed at the same time as the subtraction in function *dilateAndSubtract.*

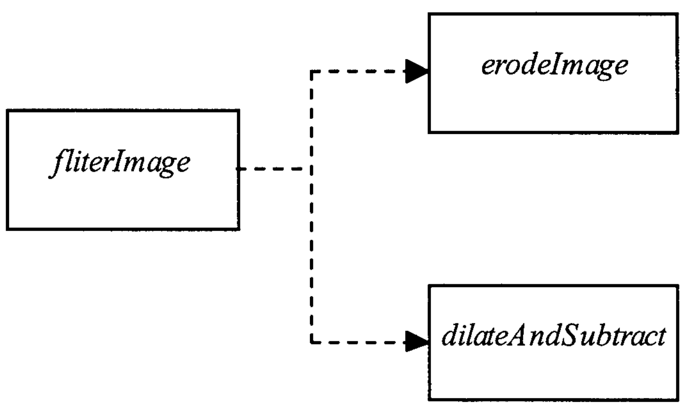


Figure B-4: Filtering Module Function Hierarchy

The function *erodeImage* performs the first step where the erosion of V is computed and the result is stored into the output buffer provided to store W. Kernel processes require two major looping constructs, one going over the image and one going over the kernel. When care is taken to chose the inner and outer loops, processing speed can be enhanced. For morphological calculations, any kernel pixel or image pixel that has a zero value requires NO OPeration at all. Since the input contains real data with short integer precision, the probability of it containing many values equal to zero is small. The kernel, however, is of type unsigned char and is highly likely have more than one zero value. To take advantage of these NOOPs and increase processing speed, loop over the kernel first and then over the image while calculating the erosion. Then, for any kernel pixel that is zero, the input image does not need to be referenced at all and processing can jump to the next kernel pixel! If the loops were in the reverse order, every pixel in the image would be accessed before discovering that a kernel pixel with a zero value required no operation. The erosion is not fully calculated until the entire outer loop over the kernel is complete. For the simple erosion, in a trade-off for speed versus memory, an implementation favoring speed wins.

After the erosion is calculated, the second step is implemented by function *dilateAndSubtract*. Here the dilation and subtraction are computed using a temporary buffer (half the number of rows of K times the number of columns of V) so that the values can be calculated and stored back into the output buffer for W. Since this step performs a calculation in-place (with the use of the temporary buffer for the overhead processing required to allow an in-place calculation) the same optimization in loop order for the erosion cannot be used. When the calculations are in-place, the entire computation must be performed for a single pixel before continuing onto the next input pixel. For this in-memory calculation on images that are very large, in a trade-off for speed versus memory, an implementation favoring speed at the expense of memory loses. Since the input images may be very large, it is more important to minimize memory in this step than to increase speed. However, for the first step, where you are guaranteed to have an input buffer for V and an output buffer for W, the optimization for speed wins (the added memory is not expensive because you are guaranteed to have it).

As discussed in Section B.3, this buffering of half the kernels length times the width of the image favors an image with a width smaller than the length. The baseline implementation exploits this advantage by transposing the input image (and the kernel) whenever the input image has more pixel columns than pixel rows.

This design is not optimal for every data set possible or for many hardware configurations. The assumptions are that the input image will be large in size and the kernels relatively small and that the hardware has limited cache and no parallelization capabilities. Then, on average, these implementation enhancements will increase the speed of execution of the filtering module.

B.4.3 <u>Region Selection Module</u>

The `Region Selection` module implements the ROI Selection component and requires for input: images V and W, along with parameters: *thresholdLevel, minArea, maxRatio,* and *selectNumber*. The output of this module is an object image, O and a list of regions in the image, *regions* (containing initial feature values). The internal parameter, image O, was chosen to be a labeled image in this implementation (see Section B.3). First the threshold is applied to the filtered image W and pixels passing the threshold are considered on target and grouped together into connected regions defined as ROIs. Then, initial features for each ROI in image W are computed and selection logic is used to choose a subset of these ROIs. Finally, the selected ROIs are labeled and stored in object image O, and the corresponding initial features are included in the list of ROI objects, *regions*. The highest level routine in this module is *selectRegions*, which provides an interface with the main line as shown in Figure B-2. Since the intermediate filtered image W is no longer needed in the sequence, the baseline implementation reuses this memory to store the labeled image O. Having the image W and image O coincide in memory saves the process a memory block the size required for an intermediate image.

Many algorithms exist to group neighboring pixels into a connected region. Two types of these connected component algorithms were considered for implementation. One uses the seed fill approach by [Heckbert] and the other is a raster scan approach described by [Lumia]. The seed fill algorithm takes a seed point and connects neighbors with the same value as the seed point to a given new value. A stack is used to store neighboring pixels of interest and processing continues until the stack is empty. This algorithm is often useful in paint utilities and was not used here because of the irregular memory access required and the possibility of the stack growing unmanageable. However, this algorithm, as well as others that exist, should be considered as a viable alternative for others who implement this benchmark.

The raster scan approach described in [Lumia] contains three variations. The first variation traverses the image line by line assigning pixels to objects and updating an equivalence table that keeps track of objects that have been merged. As the image is traversed and distinct labels are assigned to pixels to represent distinct objects, it is possible for two separate objects to become connected. Then these two connected objects are really one object and need to be merged into one object. This is achieved by associating multiple labels to one object, where one of these labels will be defined as the label for that object and the other labels will be aliases for that object. Therefore, when the end of the image is processed, each distinct object is labeled and may have any number of aliases associated to it. These aliases are stored on an equivalence table defining which labels go to which object. Thus, for large images with complex objects, the size of the equivalence table may grow to an unmanageable size. The second raster scan variation has no equivalence table and iterates on the image in two passes: one from top to bottom and the other from bottom to top. While the image is traversed a flag is set if any objects were merged. The iterations continue until an iteration occurs that does not have the flag set (where no objects were merged). This variation requires an undetermined number of passes through the image, which may be very large. The third raster scan variation is a hybrid of the other two where the image is traversed twice: once top to bottom and then from bottom to top. For each image line, an equivalence table is kept to keep track of merged objects (which is used to relabel the objects). While each image line is processed, the equivalence table keeps track of merged objects. At the end of the image line, if any objects have been merged, the merged pixels are relabeled using the data on the equivalence table. This method has a smaller equivalence table and a predetermined number of passes through the image, but may require revisiting many pixels in order to relabel objects.

For this benchmark, assuming that the targets (or objects) to be labeled are never going to be very large compared to image size or very intricate in terms of shape, the first raster scan approach has been implemented. The size of the equivalence table, however, is bounded and a relabeling procedure is performed whenever the limits of the table are reached.

The algorithm used, for the baseline implementation, to group neighboring pixels into regions is described as follows. As the image *W* is traversed, any pixel value larger than *thresholdLevel* is considered a target pixel. The image is traversed from the top row to the bottom row and from the left column to the right column. The neighbors of a target pixel (above and to the left – or neighboring pixels that have already been processed) are checked to see if any are labeled. If none of the neighbors are labeled, then the target pixel represents a new object and a new object label is used. If only one neighbor is labeled, then label the target pixel with the same label. If more than one neighbor has been labeled: if they are labeled with the same label, label the target pixel with that label; if they are labeled with different labels, then use the label from one neighbor while insuring that an alias equal to the labels from the other neighbors exist in the table (merging objects as required). The image is traversed once, processing each pixel one at a time until all the pixels have been processed and assigned a label.

This implementation does not allow an object pixel to be on the outer shell (one pixel in width) of the image. Furthermore, the pixels in this outer shell are disqualified as target pixels and are explicitly set to the background value. Since the filtering module, which precedes the region definition, contains a two-step kernel process, unless the kernel is one pixel wide, it is guaranteed that the image will have a border of undefined data (set to zero) greater than one pixel. Thus, when the image is traversed to group neighboring pixels, a faster single looping construct can be used rather than the slower traditional double looping construct (one for rows and one for columns). And since the outer shell will never have a target pixel, when neighboring pixels of a target pixel (not on the outer shell) are accessed, special cases at the boundary are avoided.

Whenever a new label is assigned to a target pixel, an object is added to the list *regions* and is assigned that label. The structure for this object, **ObjectEntry** shown in Table B-1, is used to store information about the object represented by that label. Supporting structures, **AliasEntry**, **BoundingBox**, **Point**, and **SomeFeatures**, are shown in Table B-2 through Table B-5. The **AliasEntry** structure is used to make the alias list for a given object. Values defining a bounding box around the object are stored in the **BoundingBox** structure (with **Point** structures as members), and partial initial feature calculations for the object are kept in the **SomeFeatures** structure.

Table B-1: ObjectEntry Structure Definition

type specifier	member name	comment
ObjectEntry	*nextObject	pointer to next object
ObjectEntry	*lastObject	pointer to last object
AliasEntry	*aliasList	pointer to alias list
ShortInt	labelValue	label value
BoundingBox	box	bounding box around ROI
Float	rankMetric	ranking metric
SomeFeatures	initFeatures	initial feature structure

Table B-2: AliasEntry Structure Definition

type specifier	member name	comment
AliasEntry	*nextAlias	pointer to next alias
ShortInt	aliasLabel	alias label value

Table B-3: BoundingBox Structure Definition

type specifier	member name	comment
Point	upperLeft	upper left point of bounding box
Point	lowerRight	lower right point of bounding box

Table B-4: Point Structure Definition

type specifier	member name	comment
Int	column	column coordinate of point
Int	row	row coordinate of point

Table B-5: SomeFeatures Structure Definition

type specifier	member name	comment
Float	centroidColumn	centroid column feature
Float	centroidRow	centroid row feature
Int	area	area feature
Int	perimeter	perimeter feature
Double	mean	mean feature
Double	variance	variance feature

The initial features are not dependent on spatial relationships between pixels within the same object (or ROI) and can be calculated incrementally. Thus, as each target pixel is visited and assigned a label, its contribution to the features is calculated and stored on the **ObjectEntry** structure. The centroid, mean, and variance incremental values need to be normalized once all of the pixels contributing to an object have been visited.

The output of this module, *regions*, is list of objects or a linked list of **ObjectEntry** structures. This linked list of objects may contain a linked list of aliases for each object using the **AliasEntry** structure. The label value associated to the object, a bounding box around the object, initial features for the object, and a ranking metric (used to rank the objects) completes the members on an **ObjectEntry** structure. The structure used to store the initial features, **SomeFeatures**, is used to store the incremental values composing the feature until the entire object is defined. These incremental features are changed (consisting of a normalization where appropriate) to represent the features defined in the specification – *centroid, area, perimeter, mean,* and *variance.*

If the features were not calculated incrementally, then the ROIs must be traversed after grouping pixels into ROIs to calculate the features. This would require multiple passes of the image, once to segment the pixels, and once to calculate the features. Thus, incremental calculation eliminates the need for multiple passes and increases the speed of the process.

An object entry link is formed when more than one object is found in the image where each entry on the list is a distinct object in the image. Whenever two objects are found to be connected, they are merged (associated together where one represents an alias of the other). This is achieved by deleting one of the merging objects from the object list and creating an alias with the same label as the deleted object onto the alias list for the other merging object. The list *regions* is a linked list of linked lists. One type of link is an object link and the other is an alias link. While the image is being traversed and the ROIs are being connected, many objects may be merged together. Each time a merge takes place, an entry is deleted from the object list and an alias with the same label is added to the appropriate alias list keeping track of these connections. The bounding box definition and the initial feature calculations for each object are also combined before the object is deleted from the object list and the alias is placed on the alias list. This entails consolidating information on the **ObjectEntry** structures or merging the statistics of the two objects to create statistics representative of the merged object. Once all the pixels in the image have been assigned to an ROI, then a final computation step is performed. This consolidation step is required to normalize the *centroid, mean,* and *variance* since it is now possible to calculate the total number of pixels in an ROI. The *area* feature does not require the normalization step. This consolidation step is also used to prune out objects that do not satisfy the criteria driven by *minArea* and *maxRatio*. Then the objects can be ranked in order so that the *selectNumber* objects with the highest values for the ranking metric (*mean*area)* will be retained. At the end of this module, *regions* contains a list of objects where each entry on the list represents an ROI that passes the all of the selection criteria set. The members of the **ObjectEntry** structure contain for each ROI: 1) the labels used in image *O* to label the ROI (one label plus any number of alias labels), 2) the bounding box around the ROI, 3) initial features *centroid*, *area*, *perimeter*, *mean*, and *variance*, and 4) the ranking metric (*mean*area*) to be used when the ranking selection is performed.

A function hierarchy for the `Region Selection` module is shown in Figure B-5 below. Functions are represented by bold rectangular boxes with their names written in bold italic inside the box. Function boxes that have a striped diagonal patterned background denote functions that appear more than once in the flow diagram near the functions that call them (drawn more than once for clarity).

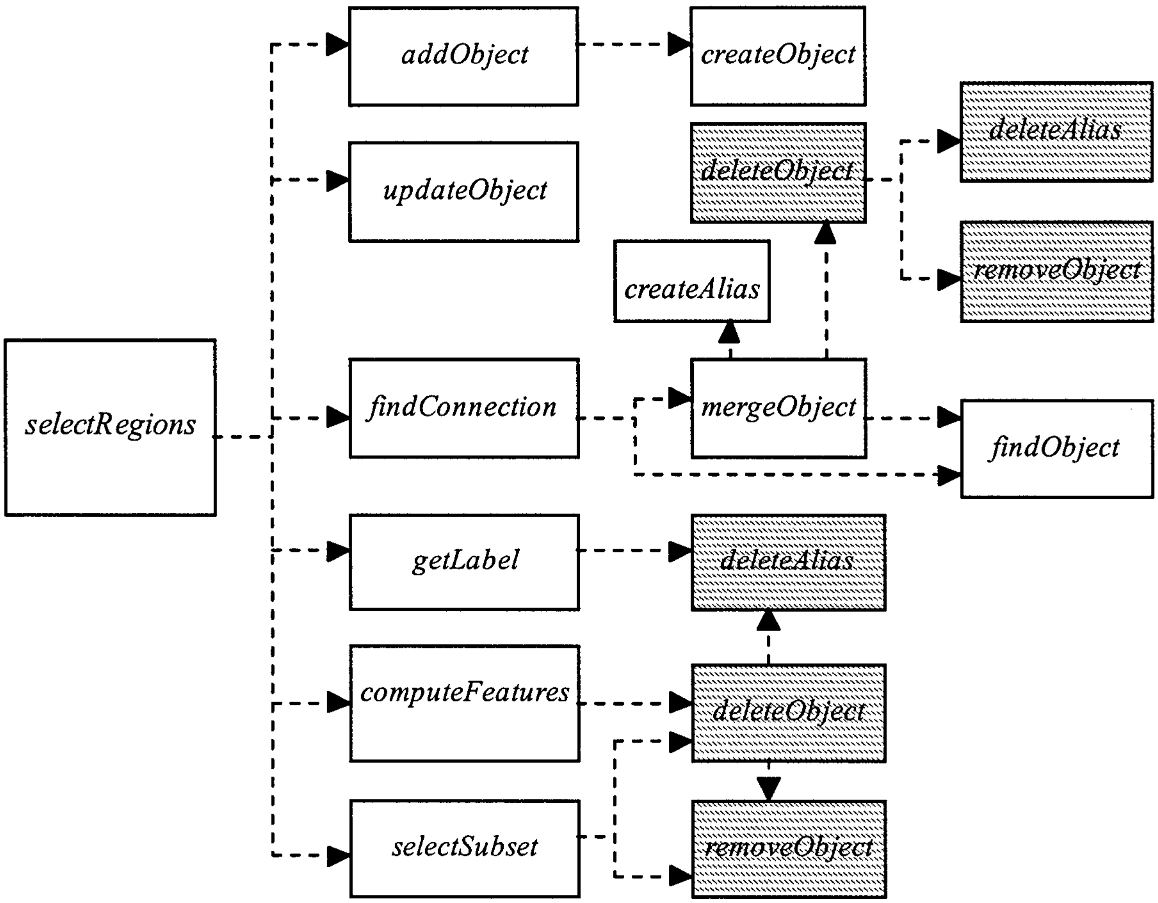


Figure B-5: Select Regions Module Function Hierarchy

The outputs from this module are the object labeled image *O* and object list *regions*. These two parameters define the shape and location of each object and store the initial features for each object. It is necessary to retain the shape and location of the object to be used later in the `Feature Extraction` module where additional features for the object are calculated. These added features are not calculated here because: 1) they are numerous (there are sixteen of them), 2) they are computationally expensive, 3) they require spatial integrity within the object so that they can not be incrementally computed, and 4) the additional features do not need to be computed on all the objects found in this module (objects that are culled using the selection criteria do not need to have additional features calculated).

A flow diagram of the highest level function in the `Select Regions` module, *select–Regions*, is depicted in Figure B-6 below. Functions are represented by bold rectangular boxes with their names written in bold italic inside the box. Function boxes that have a striped diagonal patterned background denote functions that appear more than once in the flow diagram near the functions that call them (drawn more than once for clarity). Solid arrows show the flow within the function and dotted arrows show function calls from functions called within *selectRegions*.

A list of each module called within *selectRegions* is given below. A low-level description of each module is provided in the Pseudo-code part of this document.

addObject	Add a new **ObjectEntry** with a given *label* to a list of objects.
computeFeatures	Finish computing initial features for a list of **ObjectEntries.** Normalize *centroid, mean,* and *variance*. Apply the selection criteria *minArea* and *maxRatio* to eliminate objects. Compute ranking metric (*mean* * *area*) to be used later (see Figure B-7).
createAlias	Allocates and initializes memory for an **AliasEntry** structure.
createObject	Allocates and initializes memory for an **ObjectEntry** structure
deleteAlias	Frees the memory for an **AliasEntry** structure.
deleteObject	Frees the memory for an **ObjectEntry** structure.
findConnection	Find the connection between the current pixel and its neighbors to determine what label to assign to the current pixel depending on which neighbors are labeled (see Figure B-8).
findObject	Finds the **ObjectEntry** structure associated to a label value where the object found has the given label value as its label value or an alias of its label value (see Figure B-9).
getLabel	Get a label to use for a new object's label. When necessary, recycle a used alias by relabeling the labeled image removing an alias label so that that alias label can be reused.
mergeObject	Determines whether a given *label* and given *object* need to be merged. First, if the *label* is already associated to the *object* (as its label or an alias of its label) nothing is done. Next, if the *label* is not associated to the *object*, find the object associated to the *label* and merge the two objects. Add the found object to the given *object's* alias list (remove it from the linked list of objects, create a new alias, add it to the *object's* alias list - adding its alias list to the *object's* alias list too) (see Figure B-10).
removeObject	Removes an **ObjectEntry** structure from a linked list of **Object–Entries** updating the pointer to the list of objects as required.
selectSubset	Rank the objects on the given list using the ranking metric (*mean***area*), and keep the objects with the *selectNumber* highest values (see Figure B-11).
updateObject	Incrementally update the data associated to an object's **Object-Entry** structure - the initial features and the bounding box associated to that object (one pixel's contribution).

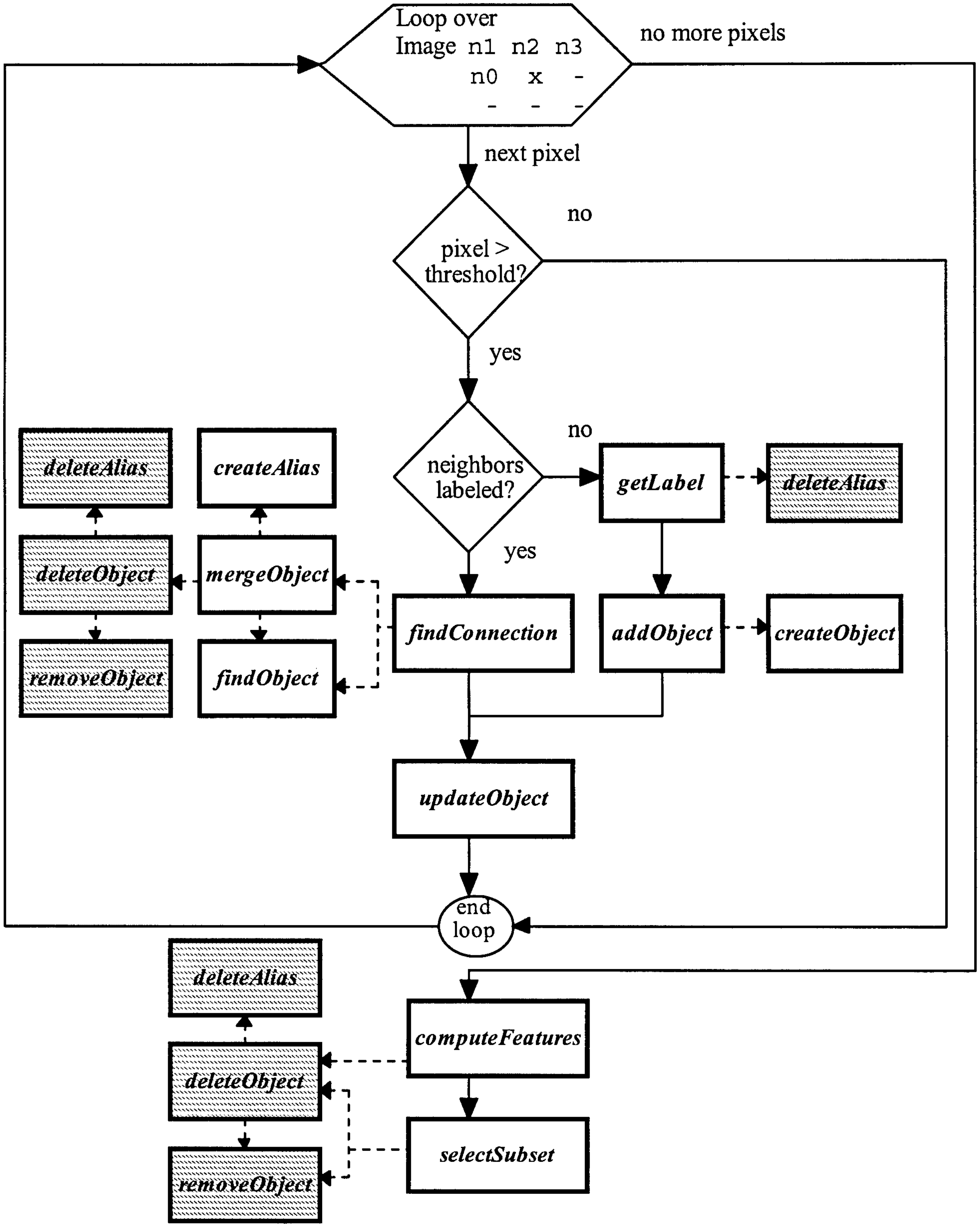


Figure B-6: Select Regions Module Flow Diagram

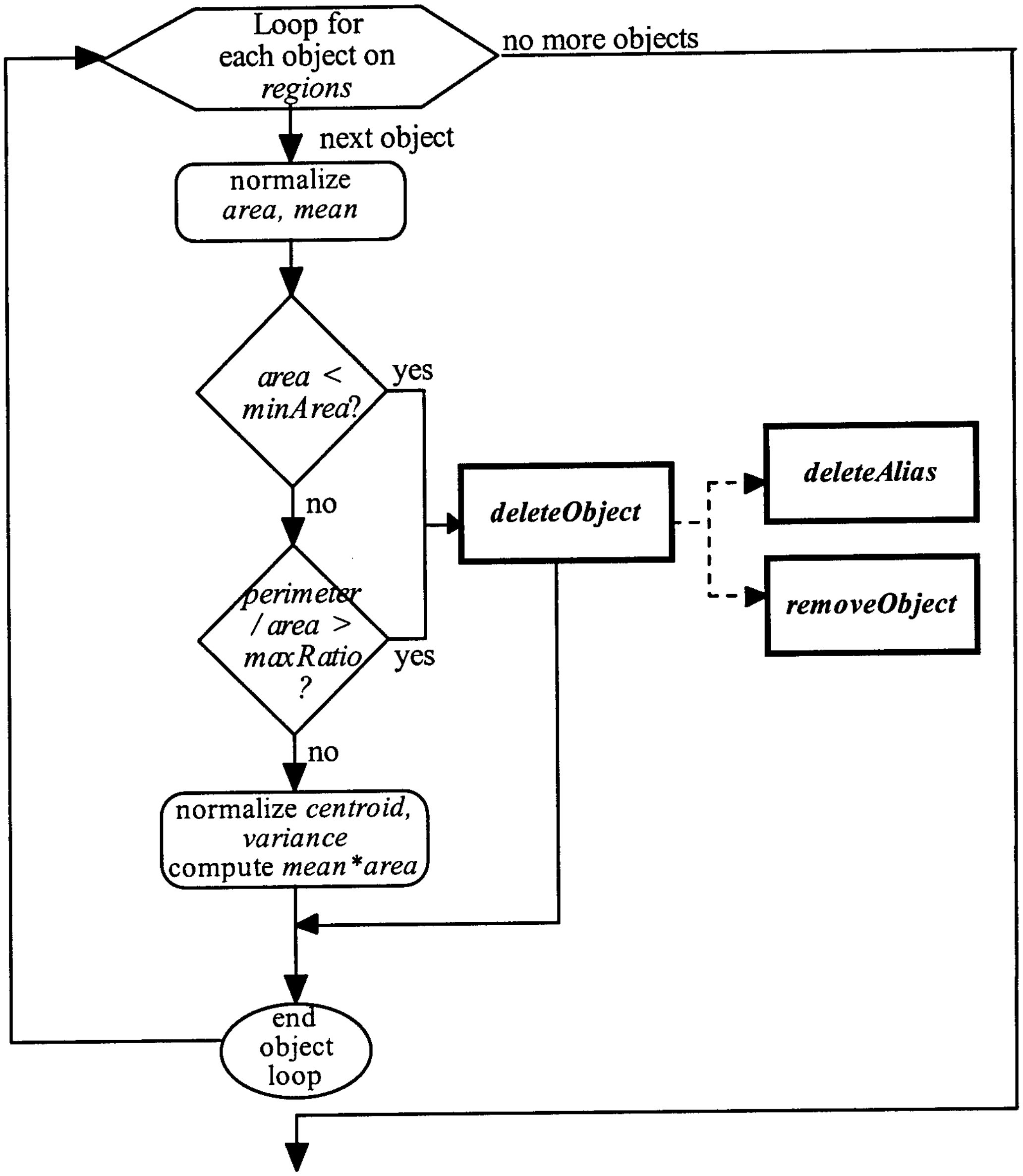
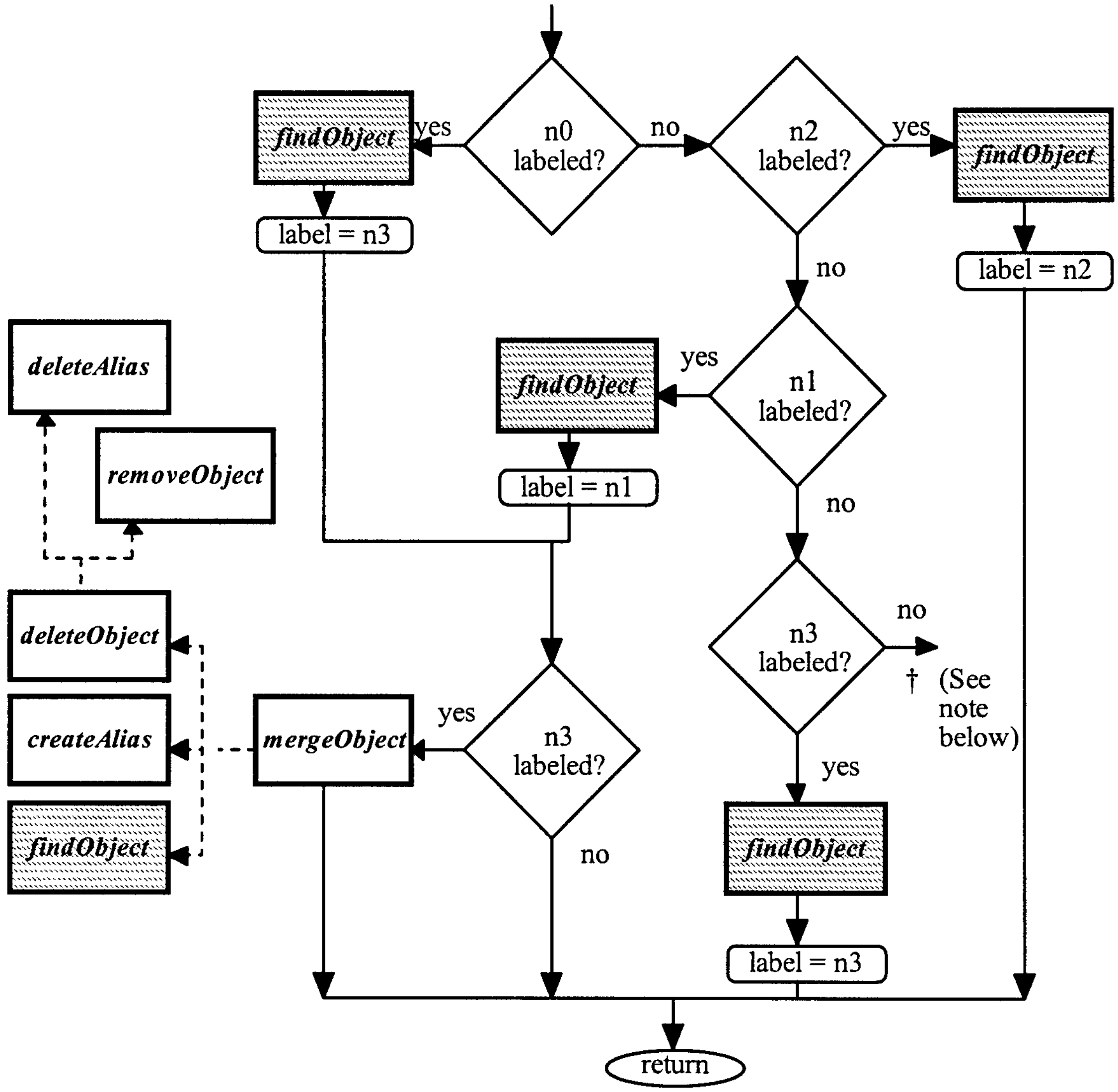


Figure B-7: *computeFeatures* Flow Diagram



Neighobor Definition:	
	n0 = left neighbor
	n1 = upper left neighbor
`n1 n2 n3`	n2 = upper neighbor
`n0  x  -`	n3 = upper right neighbor
` -  -  -`	x = pixel being processed
	- = neighbor in any state

† This function is called only when one or more of the neighbors are labeled, so the flow should never reach this point. When no neighbors are labeled, the pixel being processed is the beginning of a new object and is handled differently in the calling function (see *selectRegions*).

Figure B-8: *findConnection* Flow Diagram

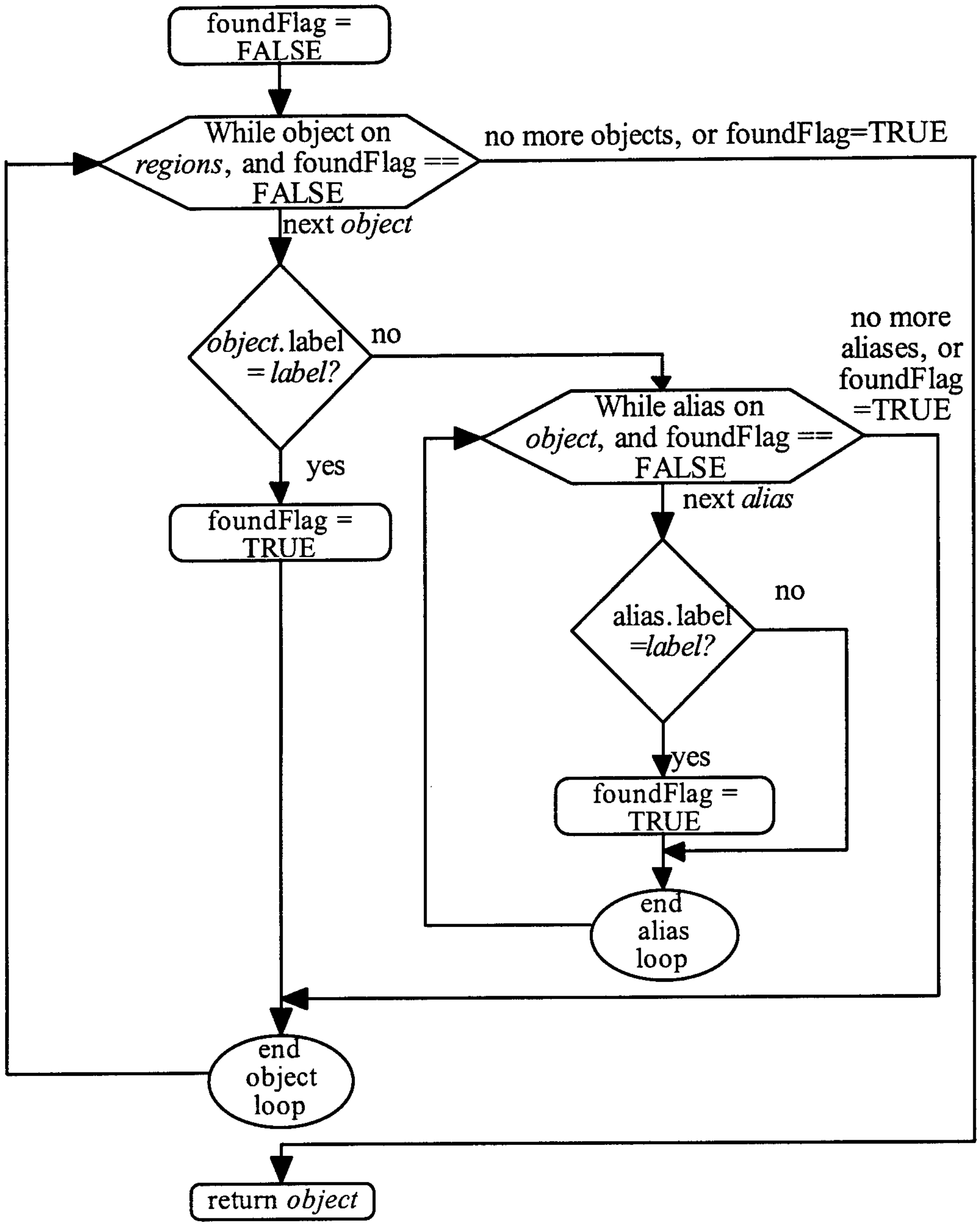


Figure B-9: *findObject* Flow Diagram

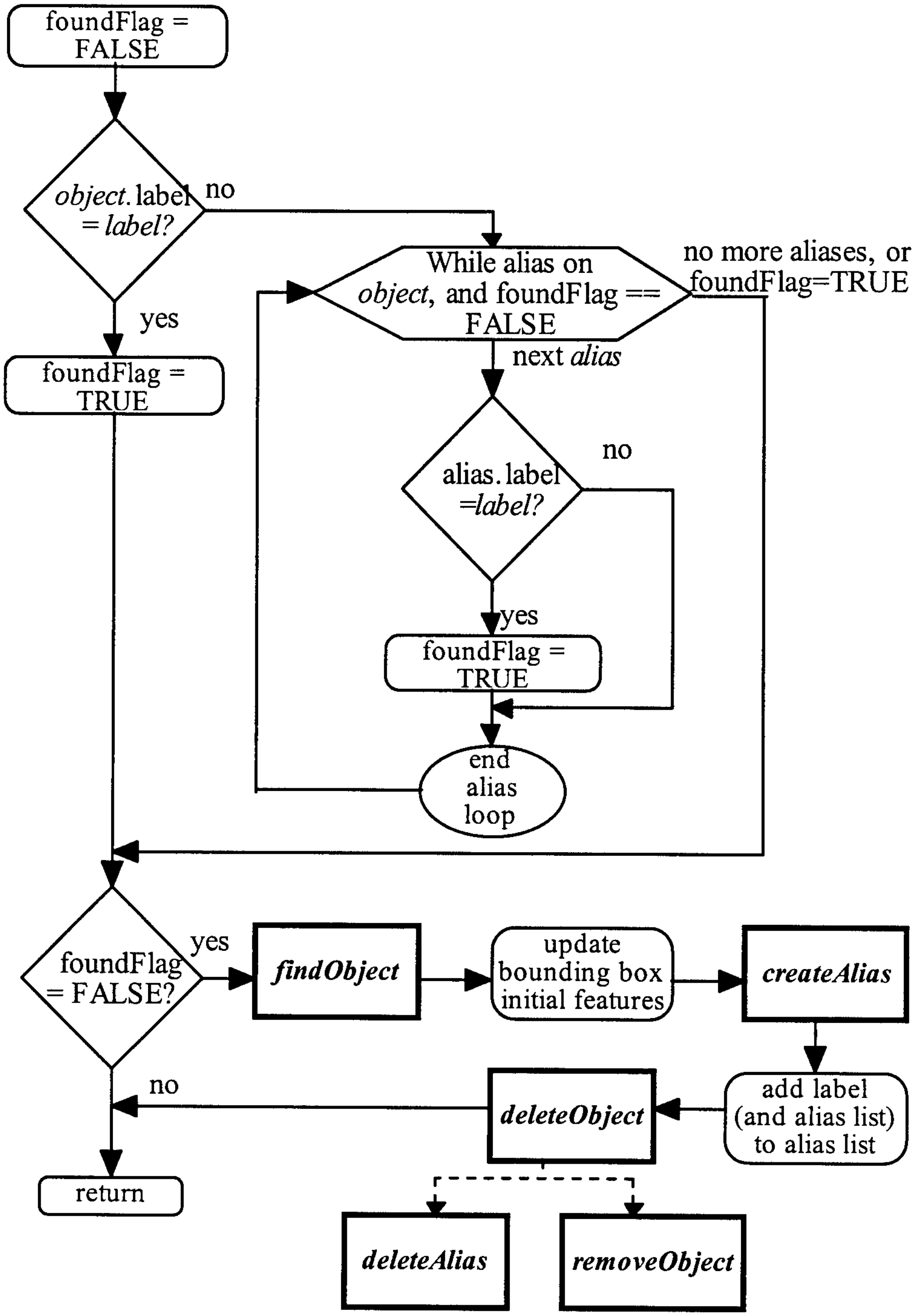


Figure B-10: ***mergeObject*** **Flow Diagram**

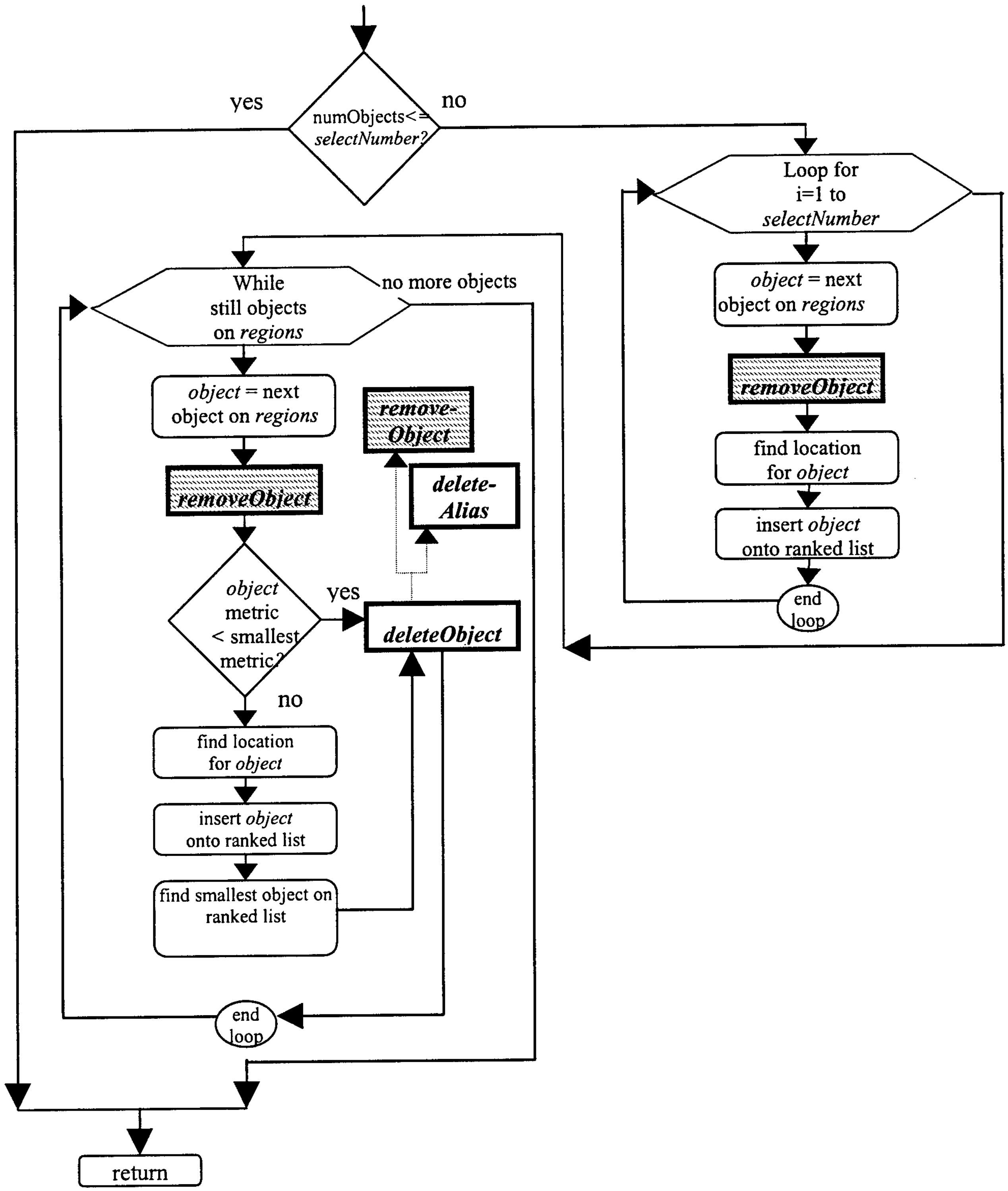


Figure B-11: *selectSubset* Flow Diagram

B.4.4 Feature Extraction Module

The `Feature Extraction` module implements the feature extraction component of the sequence. The inputs to this module are: parameters *distanceShort* and *distanceLong*, input image *V*, object image *O*, and list *regions*. The output of this module is the final output for the sequence and is the total feature list, *features,* for each selected ROI. The routine *extractFeatures* is the high level routine in this module and interfaces with *main* as shown in Figure B-2.

A function hierarchy for the `Feature Extraction` module is shown in Figure B-12 below. Functions contained in other modules, but called within this module are also included and are shaded to signify that they are functions called outside of the module.

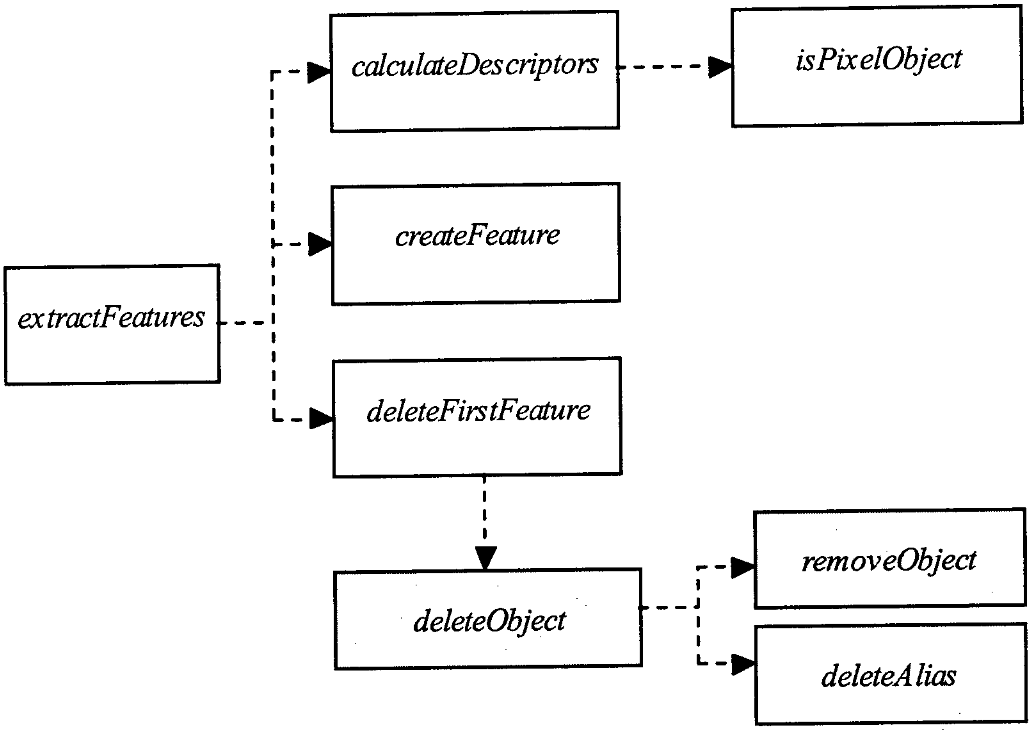


Figure B-12: Feature Extraction Module Function Hierarchy

This module goes through the list of objects *regions*, and for each object calculates the GLCM descriptors *entropy* and *energy* for each of two given distances and four defined directions (see the specification [AAEC-1]). The structure, **FeatureEntry,** shown in Table B-6 is used to store the sixteen GLCM descriptor features for each object as well as point to the **ObjectEntry** containing the other features already calculated. Since the number of features calculated in this module is large, a new structure is introduced rather than having a place holder in **ObjectEntry**. The inclusion of the GLCM descriptors in the **Object–Entry** structure was rejected since the total number of objects is expected to be large. Yet, most of these objects will be culled prior to the calculation of the descriptors.

Table B-6: FeatureEntry Structure Definition

type specifier	member name	comment
FeatureEntry	*nextFeature	pointer to next feature
ObjectEntry	*objectPtr	pointer to object entry
MoreFeatures	addFeatures	additional feature structure

Supporting structures, **Descriptors**, **Angles**, and **MoreFeatures**, are shown in Table B-7 through Table B-9. The **Descriptors** structure contains the GLCM *entropy* and *energy* feature values. The **Angles** structure holds descriptor values for the four angles required, 0, 45, 90, and 135 degrees. And the **MoreFeatures** structure consists of **Angles** for both distances, *distanceShort* and *distanceLong*, to contain a total of sixteen additional features for each object.

Table B-7: Descriptors Structure Definition

type specifier	member name	comment
Float	entropy	GLCM entropy
Float	energy	GLCM energy

Table B-8: Angles Structure Definition

type specifier	member name	comment
Descriptors	deg0	features for 0 degrees
Descriptors	deg45	features for 45 degrees
Descriptors	deg90	features for 90 degrees
Descriptors	deg135	features for 135 degrees

Table B-9: MoreFeatures Structure Definition

type specifier	member name	comment
Angles	distShort	features for short distance
Angles	distLong	features for long distance

The highest level function in this module is *extractFeatures* whose flow is depicted in Figure B-13 below. Functions are represented by bold rectangular boxes with their names written in bold italic inside the box. Functions contained in other modules, but called within this module are also included and are shaded to signify that they are functions called outside of the module. Function boxes that have a striped diagonal patterned background denote functions that appear more than once in the flow diagram near the functions that call them (drawn more than once for clarity). Solid arrows show the flow within the function and dotted arrows show function calls from functions called within *extractFeatures.*

A list of each module called within *extractFeatures* is given below. A low-level description is provided in the Pseudo-code part of this document.

calculateDescriptors	Calculate the GLCM *energy* and *entropy* for an object (see Figure B-14).
createFeature	Allocates and initializes memory for a **FeatureEntry** structure.
deleteFirstFeature	Frees the memory for the first **FeatureEntry** structure on the list.
isPixelObject	Determines whether a given pixel location is part of an object (see Figure B-15).

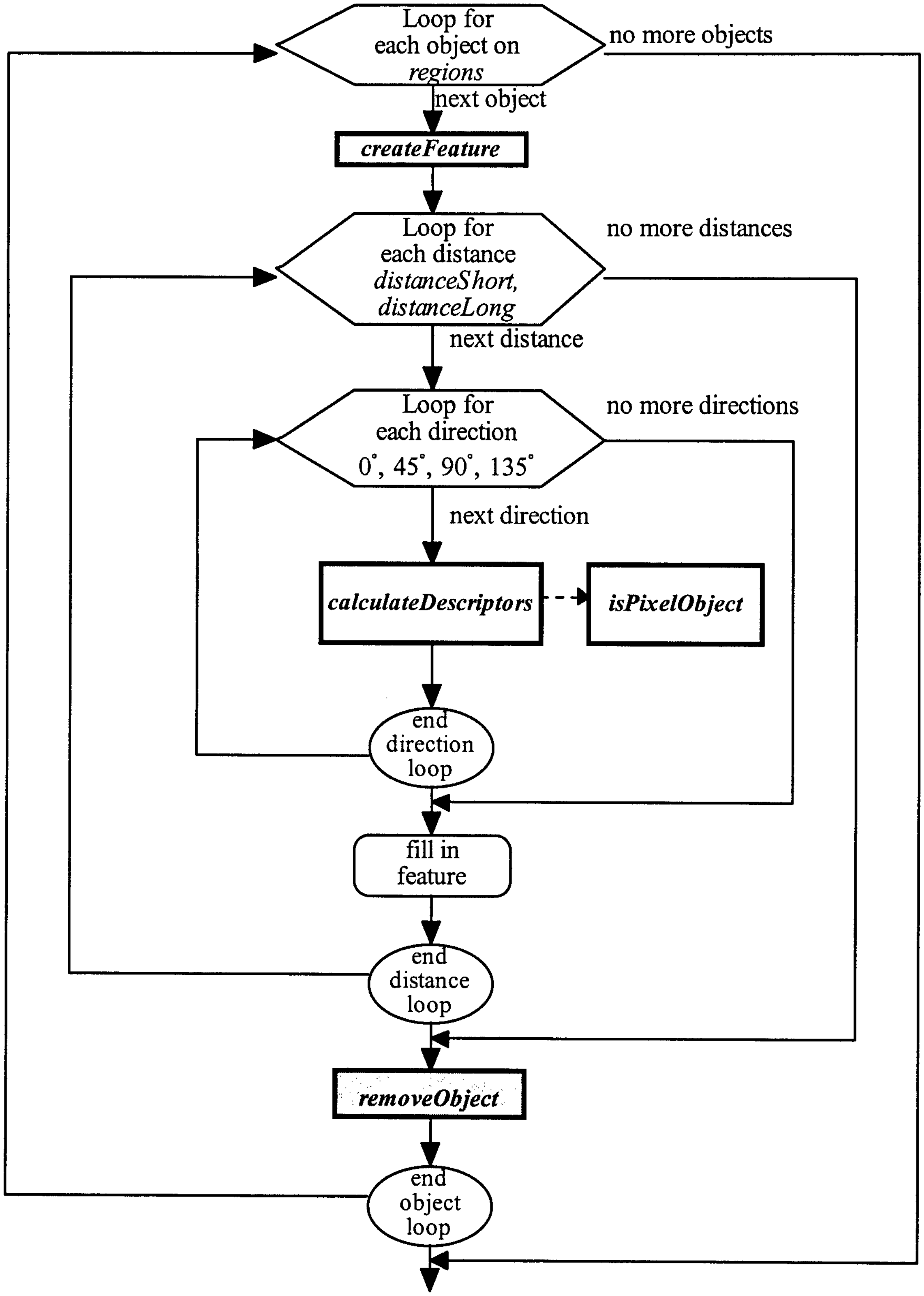


Figure B-13: Feature Extraction Module Flow Diagram

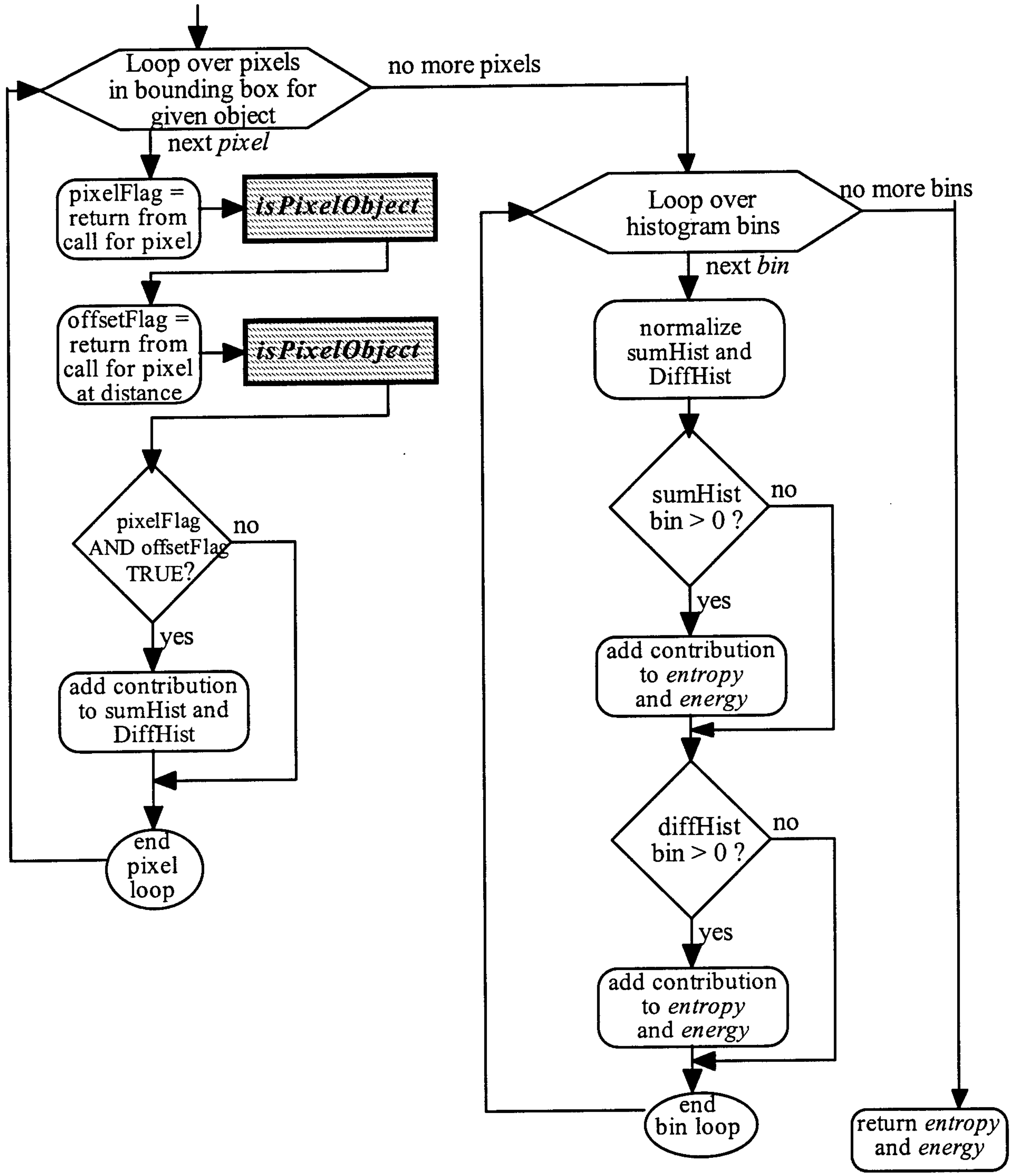


Figure B-14: *calculateDescriptors* Flow Chart

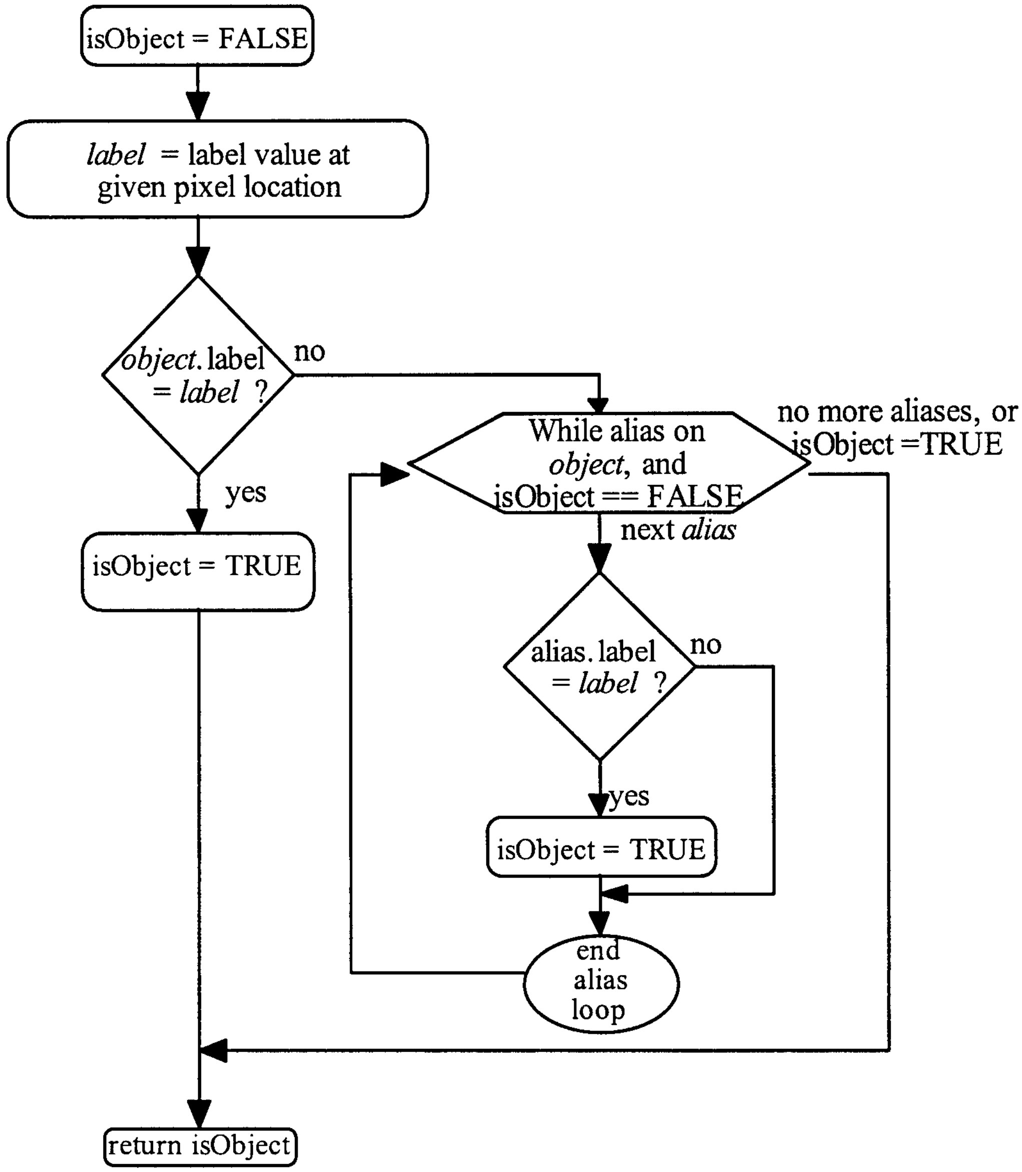


Figure B-15: *isPixelObject* Flow Chart

B.5 REFERENCES

[AAEC-1] "Data-Intensive Systems Benchmark Suite," MDA972-97-C 0025, Atlantic Aerospace Electronics Corp., Waltham, MA 02451, Spring 1999.

[AAEC-2] "DIS Benchmark C Style Guide: C Style and Coding Standards for the DIS Benchmark Suite," MDA972-97-C 0025, Atlantic Aerospace Electronics Corp., Waltham, MA 02451, December 28, 1998.

[Heckbert] Heckbert, Paul S., (Glassner, A. - editor), *Graphic Gems*, Academic Press, 1990, pp. 275, 721.

[Lumia] Lumia, R., Shapiro, L., and Zuniga, O., "A New Connected Components Algorithm for Virtual Memory Computers," *Computer Vision, Graphics, and Image Processing,* **22**, pp. 287-300, 1983

B.6 PSEUDO-CODE

This part of the document lists the pseudo-code for the baseline application. The first section contains a listing of the variable types and command structures. The sections that follow contain each module of the sequence: `Input & Output`, `Filtering`, `Region Selection`, and `Feature Extraction`. Details for each module are provided in the following subsections starting with the highest level routine for the module, followed by lower-level functions in alphabetical order. The routine descriptions and comments are provided along with the pseudo-code.

B.7 COMMON TYPES AND STRUCTURES

```
typedef char            Char;
typedef unsigned char   UChar;
typedef long int        Int;
typedef short int       ShortInt;
typedef float           Float;
typedef double          Double;

typedef Char            Boolean;

/*
 * Image structure for images
 */
typedef struct {
Int       numColumns; /* number of columns in image */
Int       numRows;    /* number of rows in image */
ShortInt *data;       /* data pointer */
} ShortIntImage;

/*
 * Image structure for kernels
 */
typedef struct {
Int   numColumns; /* number of columns in image */
Int   numRows;    /* number of rows in image */
UChar *data;      /* data pointer */
} UCharImage;

/*
 * Structure for initial features
 */
typedef struct {
   Float       centroidColumn;   /* centroid column feature */
   Float       centroidRow;      /* centroid row feature */
   Int         area;             /* area feature */
   Int         perimeter;        /* perimeter feature */
   Double      mean;             /* mean feature */
   Double      variance;         /* variance feature */
} SomeFeatures;

/*
 * Structures for additional features
 */
```

```
typedef struct {
   Float entropy;     /* GLCM entropy */
   Float energy;      /* GLCM energy */
} Descriptors;
typedef struct {
   Descriptors deg0;    /* features for   0 degrees */
   Descriptors deg45;   /* features for  45 degrees */
   Descriptors deg90;   /* features for  90 degrees */
   Descriptors deg135;  /* features for 135 degrees */
} Angles;
typedef struct {
   Angles   distShort;  /* features for short distance value */
   Angles   distLong;   /* features for long distance value */
} MoreFeatures;

/*
 * Structures for ROI entries on equivalence table/list with features
 */
typedef struct {
   Int      column;  /* column coordinate of point */
   Int      row;     /* row coordinate of point */
} Point;
typedef struct {
   Point    upperLeft;  /* upper left point of bounding box */
   Point    lowerRight; /* lower right point of bounding box */
} BoundingBox;
typedef struct AliasType {
   AliasType      *nextAlias;    /* pointer to next alias */
   ShortInt       aliasLabel;    /* alias label value */
} AliasEntry;
typedef struct ObjectType{
   ObjectType     *nextObject;   /* pointer to next object */
   ObjectType     *lastObject;   /* pointer to last object */
   AliasEntry     *aliasList;    /* pointer to alias list */
   ShortInt       labelValue;    /* label value */
   BoundingBox    box;           /* bounding box around ROI */
   Float          rankMetric;    /* ranking metric */
   SomeFeatures   initFeatures;  /* initial features */
} ObjectEntry;
typedef struct FeatureType{
   FeatureType    *nextFeature;  /* pointer to next ROIs features */
   ObjectEntry    *objectPtr;    /* pointer to object pointer (to
                                  * access the initial features)
                                  */
   MoreFeatures   addFeatures;   /* additional features */
} FeatureEntry;
```

B.8 INPUT & OUTPUT MODULE

B.8.1 main

Name:	main
Input:	input file name (option w/default being stdin)
Output:	feature output file (option w/default being stdout)

	metrics output file (option w/default being stderr)
	error output (directed to stderr)
Description:	This program executes the Image Understanding sequence. The main routine which invokes the sequence calling the highest level functions in the other modules, interfaces between the I/O, and calculates the metrics. The control for the implementation is at this level and includes allocating and freeing memory for intermediate results when necessary. Time stamps are taken to provide the metrics specified in the specification [AAEC-1]. To minimize the space required, the routine *selectRegions* overwrites its input buffer for image *W* to store output labeled image *O*. Memory buffers are freed as soon as they are no longer required.
Calls:	createShortIntImage() errorMessage() extractFeatures() filterImage() flushErrorMessage() freeShortIntImage() freeUCharImage() readInput() selectRegions() writeOutput()
(system)	difftime() time()

```
{ /* begin main */
   Get command-line Int, argc          /* # of line arguments      */
   Get command-line arguments, argv /* command-line arguments  */
   Define Char, *inputFile             /* input file name          */
   Define Char, *outputFile            /* output file name         */
   Define Char, *metricFile            /* metric file name         */
   Define Int, error                   /* error flag for routines */
   Define UCharImage **kernelK         /* pointer to kernel image K */
   Define ShortIntImage **imageV       /* pointer to input image V */
   Define ShortIntImage **imageW       /* pointer to image W
                                        * (reused to store image O
                                        */
   Define Boolean transposeFlag        /* flag for transpose */

   /*
    * Get filenames or set variables to NULL to obtain defaults for
    * input, output, and metric files
    */
   startTime = time();
   set up input either files specified by user
      or stdin, stdout, and stderr

   /*
    * Read input file
    */
   Call error = readInput(inputFile, &imageV, &kernelK, &thresholdLevel,
                          &minArea, &maxRatio, &selectNumber,
                          &distanceShort, &distanceLong, &transposeFlag)
```

```
If error != READ_INPUT_SUCCESS Then
   errorMessage("main: ");
   flush error buffer
   Return          /* FATAL: unable to open files */
EndIf

imageW = createShortIntImage( imageV->numColumns, imageV->numRows )
If imageW == NULL Then
   freeUCharImage(&kernelK)
   freeShortIntImage(&imageV)
   errorMessage("main: Error creating buffer for filtered image\n");
   flush error buffer
   Return          /* FATAL: unable to allocate memory */
EndIf

/*
 * Morphological Filter component
 */
Set markTime = time()
Call error = filterImage( kernelK, imageV, imageW )
If error != FILTER_IMAGE_SUCCESS Then
   freeUCharImage(&kernelK)
   freeShortIntImage(&imageV)
   freeShortIntImage(&imageW)
   errorMessage("main: Error in Filtering component - EXITTING\n");
   flush error buffer
   Return          /* ERROR: filter image unsuccessful */
EndIf
calculate filterTime
freeUCharImage(&kernelK)

/*
 * ROI Selection component
 */
Set markTime = time()
Call error = selectRegions(imageV, thresholdLevel, minArea,
                           maxRatio, selectNumber, imageW, &regions )
If error != SELECT_REGIONS_SUCCESS Then
   freeShortIntImage(&imageV)
   freeShortIntImage(&imageW)    /* which also frees imageO */
   loop for each object on regions
      deleteObject(object)
   endloop
   errorMessage(
      "main: Error in Region Selection component - EXITTING\n");
   flush error buffer
   Return          /* ERROR: select regions unsuccessful */
EndIf
calculate selectRegionsTime

/*
 * Feature Extraction component
 */
Set markTime = time()
Call error = extractFeatures(imageV, imageW, regions, distanceShort,
                             distanceLong, &features)
If error != EXTRACT_FEATURES_SUCCESS Then
```

```
      freeShortIntImage(&imageV)
      freeShortIntImage(&imageW)     /* which also frees imageO */
      loop for each item on features
         deleteFirstFeature(item)
      endloop
      errorMessage(
         "main: Error in Region Selection component - EXITTING\n");
      flush error buffer
      Return          /* ERROR: extract features unsuccessful */
   EndIf
   calculate extractFeaturesTime
   freeShortIntImage(&imageV)
   freeShortIntImage(&imageW) /* which also frees imageO */

   /*
    * Write output file
    */
   Call error = writeOutput( outputFile, transposeFlag, &features )
   If error != WRITE_OUTPUT_SUCCESS Then
      Return          /* FATAL: unable to write output */
   EndIf
   calculate totalTime

   /*
    * Write metric file
    */
   Open metricFile
   report filterTime, selectRegionsTime, extractFeaturesTime, totalTime

   /*
    * Clean up before Return
    */
   Return SUCCESS
```

B.8.2 createShortIntImage

Name:	createShortIntImage
Input:	numColumns, numRows
Comment:	This routine allocates memory to store an image and fills in the shortIntImage structure except for the image data. If there is an error allocating memory, than a NULL pointer is returned. Assumptions: the image dimensions given are positive and greater than zero.
Calls:	errorMessage()
(system)	malloc() free()
Return:	shortIntImage structure pointer (partially filled in – no data – and allocated) NULL (error allocating memory for image/image structure)

```
{ /* begin createShortIntImage */
   assert(dimensions >= 0)

   allocate memory for shortIntImage structure
   if error allocating memory
      call errorMessage(
```

```
            "createShortIntImage: Error allocating memory.\n");
      return(NULL)
   allocate memory for image data
   if error allocating memory
      free memory for shortIntImage structure
      call errorMessage(
            "createShortIntImage: Error allocating memory.\n");
      return(NULL)
   initialize members in shortIntImage structure (except for data)
   return(shortIntImage)
} /* end createShortIntImage */
```

B.8.3 createUCharImage

Name:	createUCharImage
Input:	image dimensions
Comment:	This routine allocates memory to store an image and fills in the uCharImage structure except for the image data. If there is an error allocating memory, than a NULL pointer is returned. Assumptions: the image dimensions given are positive and greater than zero.
Calls:	
(system)	malloc() free()
Return:	uCharImage structure pointer (partially filled in – no data – and allocated) NULL (error allocating memory for image/image structure)

```
{ /* begin createUCharImage */
   assert(dimensions >= 0)

   allocate memory for uCharImage structure
   if error allocating memory
      call errorMessage("createUCharImage: Error allocating memory.\n");
      return(NULL)
   allocate memory for image data
   if error allocating memory
      free memory for uCharImage structure
      call errorMessage("createUCharImage: Error allocating memory.\n");
      return(NULL)
   initialize members in uCharImage structure (except for data)
   return(uCharImage)
} /* end createUCharImage */
```

B.8.4 freeShortIntImage

Name:	freeShortIntImage
Input:	shortIntImage structure pointer
Comment:	This routine frees memory to associated to a shortIntImage structure. It is assumed that the pointer being passed is a non-NULL valid image structure pointer.
Calls:	

(system)	free()
Return:	none

```
{ /* begin freeShortIntImage */
   assert(image pointer not null)

   free image data
   free shortIntImage structure
   set image pointer to NULL

   return
} /* end freeShortIntImage */
```

B.8.5 freeUCharImage

Name:	freeUCharImage
Input:	uchar image structure pointer
Comment:	This routine frees memory to associated to a uCharImage structure. It is assumed that the pointer being passed is a non-NULL valid image structure pointer.
Calls:	
(system)	free()
Return:	none

```
{ /* begin freeUCharImage */
   assert(image pointer not null)

   free image data
   free shortIntImage structure
   set image pointer to NULL

} /* end freeUCharImage */
```

B.8.6 readInput

Name:	readInput
Input:	inputFileName
Output:	*V* image, *K* kernel, thresholdLevel, minArea, maxRatio, selectNumber, distanceShort, distanceLong, transposeFlag
Description:	Open input file and read in the input kernel, image, and parameters. The input file can be a disk file or the input stream *stdin*. Pass the *transposeFlag* parameter from the routine reading in the image to the routine reading in the kernel and to the main routine. If the input image is transposed, then the kernel also is transposed, and the output must be transposed. Thus, the flag *transposeFlag,* must be retained and passed to the routines affected. Validity checks are performed on the input. The following cases are invalid and cause an error return: 1) a kernel *K* dimension is smaller than image *V* dimension 2) kernel dimension is not odd 3) no pixels in kernel are greater than zero 4) *minArea* < 0 5) *minArea* >= total area of input image *V*

6) *maxRatio* <= 0
7) *selectNumber* <= 0
8) *selectNumber* >= 2^{16} - 2 = 65534
9) *distanceShort* <= 0
10) *distanceShort* >= number columns, number rows in *V*
11) *distanceLong* <= 0
12) *distanceLong* >= number columns, number rows in *V*
Close the input file on the return.

Calls: errorMessage()
readUCharImage()
readShortIntImage()
(system) fclose()
fopen ()

Return: READ_INPUT_SUCCESS
READ_INPUT_ALLOC_ERROR
READ_INPUT_INVALID_INPUT_PARAMETER
READ_INPUT_INVALID_KERNEL
READ_INPUT_KERNEL_NOT_ODD
READ_INPUT_KERNEL_TOO_LARGE
READ_INPUT_OPEN_FILE_ERROR
READ_INPUT_READ_FILE_ERROR
READ_INPUT_CLOSE_FILE_ERROR

```
{ { /* begin readInput */
   open file (or stdin if no input filename given)
   if error
      call errorMessage("readinput: error opening input file\n")
      return READ_INPUT_OPEN_FILE_ERROR

   /*
    * Read in input image V, transposing and setting transposeFlag
    * if the number of columns > number of rows.
    */
   call readShortIntImage to get imageV and transposeFlag
   if return is READ_SHORT_INT_IMAGE_ALLOC_ERROR then
      call errorMessage("readinput: ")
      close input file
      return READ_INPUT_ALLOC_ERROR
   elseif return != READ_SHORT_INT_IMAGE_SUCCESS then
      call errorMessage("readinput: ")
      close input file
      return READ_INPUT_READ_FILE_ERROR

   /*
    * Read in kernel image K, transposing if transposeFlag is set.
    */
   call readUCharImage passing transposeFlag to get K
   if return is READ_UCHAR_IMAGE_ALLOC_ERROR then
      call errorMessage("readinput: ")
      freeShortIntImage(&imageV)
      close input file
      return READ_INPUT_ALLOC_ERROR
```

```
elseif return != READ_UCHAR_IMAGE_SUCCESS then
   call errorMessage("readinput: ")
   freeShortIntImage(&imageV)
   close input file
   return READ_INPUT_READ_FILE_ERROR

/*
 * Check validity of kernel:
 * 1) verify kernel is smaller than image
 * 2) verify kernel has dimensions that are odd
 * 3) make sure kernel has at least one pixel > 0
 */
if ((kernel numColumns > input numColumns) or
    (kernel numRows > input numRows))
   call errorMessage("readinput: kernel too large\n")
   freeShortIntImage(&imageV)
   freeUCharImage(&kernelK)
   close input file
   return READ_INPUT_KERNEL_TOO_LARGE
if ((kernel numColumns not odd) or
    (kernel numRows not odd))
   call errorMessage("readinput: kernel has even dimension\n")
   freeShortIntImage(&imageV)
   freeUCharImage(&kernelK)
   close input file
   return READ_INPUT_KERNEL_NOT_ODD
pixelCount = 0
loop over pixels
   if kernel(pixel) > 0
      pixelCount += 1
end /* loop */
if pixelCount == 0
   call errorMessage("readinput: kernel has no pixels != 0\n")
   freeShortIntImage(&imageV)
   freeUCharImage(&kernelK)
   close input file
   return READ_INPUT_INVALID_KERNEL

/*
 * Read in the input parameters.
 */
read inputs: thresholdLevel, minArea, maxRatio, selectNumber,
          distanceShort, and distanceLong
if error reading in a parameter
   call errorMessage("readinput: error reading input parameters\n")
   freeShortIntImage(&imageV)
   freeUCharImage(&kernelK)
   close input file
   return READ_INPUT_READ_FILE_ERROR

/*
 * Check validity of input parameters:
 * 4) minArea >= 0
 * 5) maxRatio > 0
 * 6) selectNumber > 0
 * 7) distanceShort > 0
 * 8) distanceLong > 0
```

```
  * 9) minArea < total area of input image V
  * 10) distanceShort < number columns, number rows in V
  * 11) distanceLong < number columns, number rows in V
  * 12) selectNumber < 2^16 - 2
  */
 if ((minArea < 0) or
     (maxRatio <= 0) or
     (selectNumber <= 0) or
     (distanceShort <= 0) or
     (distanceLong <= 0) or
     (minArea >= (numColumns in V) * (numRows in V)) or
     (distanceShort >= min[(numColumns in V),(numRows in V)]) or
     (distanceLong >= min[(numColumns in V),(numRows in V)]) or
     (selectNumber >= (2^16 - 2)))
   call errorMessage("readinput: error invalid input parameter\n")
   freeShortIntImage(&imageV)
   freeUCharImage(&kernelK)
   close input file
   return READ_INPUT_INVALID_INPUT_PARAMETER

 close input file
 if error closing input file
   call errorMessage("readinput: error closing input file\n")
   freeShortIntImage(&imageV)
   freeUCharImage(&kernelK)
   return READ_INPUT_CLOSE_FILE_ERROR

 return READ_INPUT_SUCCESS
} /* end readInput */
```

B.8.7 readShortIntImage

Name:	readShortIntImage
Input:	file pointer to input file
Output:	ShortInt image structure filled in and allocated, transposeFlag
Comment:	This routine accesses the input file, calls a routine to allocate memory to store an image and stores the short integer values in this memory. The file pointer is assumed to be open and is left open. If the input image has more columns than rows, the image is read in and transposed as it's stored into the image buffer, and the *transposeFlag* is set to TRUE. If the input image has a number of columns that is equal or less than the number of rows, the image is read in stored into the image buffer in the same orientation as provided in the input file, and the *transposeFlag* is set to FALSE. An error will be returned if: there is a problem allocating memory for the image structure, there is an error reading in the data, or the dimensions of the image are negative or equal to zero. There is also an error if the data being read in is outside of the legal data values set by DATA_LOWER_LIMIT and DATA_UPPER_LIMIT. These two values are -/+ 16383 and guarantee that the processing performed by the sequence will yield a value that will be in the dynamic range of a short integer as stipulated in the benchmark specification.
Calls:	createShortIntImage()

	freeShortIntImage()
(system)	fread()
Return:	READ_SHORT_INT_IMAGE_SUCCESS
	READ_SHORT_INT_IMAGE_ALLOC_ERROR READ_SHORT_INT_IMAGE_INVALID_DIMENSIONS READ_SHORT_INT_IMAGE_READ_ERROR READ_SHORT_INT_IMAGE_INVALID_DATA

```
{ /* begin readShortIntImage */
   /* Use file pointer that is already opened. */

   *shortIntImage = NULL   /* initialize pointer return */
   *transposeFlag = FALSE  /* initialize transposeFlag to false */

   /*
    * Read in the input image V, first read in dimensions, if the
    * number of columns and number of rows > 0 (valid dimensions), and
    * number of columns > number of rows, then transpose the image
    * as it's being read in.  Create an image structure to store
    * the image dimensions and data.
    */
   read in dimensions of image
   if error reading in dimensions
      call errorMessage("readShortIntImage:error reading dimensions\n")
      return READ_SHORT_INT_IMAGE_READ_ERROR
   if (dimensions =< 0)
      call errorMessage("readShortIntImage:error invalid dimensions\n")
      return READ_SHORT_INT_IMAGE_INVALID_DIMENSIONS
   if (number of columns > number of rows) then {
      set transposeFlag
      switch number of columns and rows
   } /* end transpose case */
   *shortIntImage = call createShortIntImage
   if *shortIntImage == NULL
      call errorMessage("readShortIntImage: ")
      return READ_SHORT_INT_IMAGE_ALLOC_ERROR
   if (transposeFlag == TRUE)
      read in image data storing the transpose
   else
      read in image data
   end /* if else */
   if data is outside of range (DATA_LOWER_LIMIT,DATA_UPPER_LIMIT)
      call freeShortIntImage(shortIntImage)
      call errorMessage("readShortIntImage: error invalid data\n")
      return READ_SHORT_INT_IMAGE_INVALID_DATA
   if error reading in data
      call freeShortIntImage(shortIntImage)
      call errorMessage("readShortIntImage: error reading data\n")
      return READ_SHORT_INT_IMAGE_READ_ERROR

   return READ_SHORT_INT_IMAGE_SUCCESS
} /* end readShortIntImage */
```

B.8.8 readUCharImage

Name:	readUCharImage
Input:	file pointer to input file, *transposeFlag*
Output:	UChar image structure filled in and allocated
Comment:	This routine accesses the input file, calls a routine to allocate memory to store an image and stores the positive integer values in this memory. The file pointer is assumed to be open and is left open. If the *transposeFlag* is set to TRUE, the kernel is read in and transposed as it's stored into the image buffer. If the *transposeFlag* is not set, the kernel is read in stored into the image buffer in the same orientation as provided in the input file. An error will be returned if: there is a problem allocating memory for the image structure, there is an error reading in the data, or the dimensions of the image are negative or equal to zero.
Calls:	createUCharImage() freeUCharImage()
(system)	fread()
Return:	READ_UCHAR_IMAGE_SUCCESS READ_UCHAR_IMAGE_ALLOC_ERROR READ_UCHAR_IMAGE_INVALID_DIMENSIONS READ_UCHAR_IMAGE_READ_ERROR

```
{ /* begin readUCharImage */
   /* Use file pointer that is already opened. */

   *uCharImage = NULL /* initialize pointer return */

   /*
    * Read in the input kernel, first read in dimensions, if the
    * number of columns and number of rows > 0 (valid dimensions),
    * then read in kernel (transposing if transposeFlag is set).
    * Create an image structure to store the kernel dimensions and data.
    */
   read in dimensions of image
   if error reading in dimensions
      call errorMessage("readUCharImage:error reading dimensions\n")
      return READ_UCHAR_IMAGE_READ_ERROR
   if (dimensions =< 0)
      call errorMessage("readUCharImage:error invalid dimensions\n")
      return READ_UCHAR_IMAGE_INVALID_DIMENSIONS
   if (transposeFlag == TRUE) then {
      switch number of columns and rows
   } /* end transpose case */
   *uCharImage = call createUCharImage
   if *uCharImage == NULL
      call errorMessage("readUCharImage: ")
      return READ_UCHAR_IMAGE_ALLOC_ERROR
   if (transposeFlag == TRUE)
      read in image data storing the transpose
   else
      read in image data
   end /* if else */
```

```
   if error reading in data
      call freeUCharImage(&uCharImage)
      call errorMessage("readUCharImage: error reading data\n")
      return READ_UCHAR_IMAGE_READ_ERROR

   return READ_UCHAR_IMAGE_SUCCESS
} /* end readUCharImage */
```

B.8.9 writeOutput

Name:	writeOutput
Input:	outputFileName, transposeFlag, features
Output:	The output file created with the features written to the file in ascii format as specified in the spec.
Description:	Open output file and write out the output contained in the features linked list. If the transposeFlag is set, then "transpose" the feature values to reflect features that are valid for input images that are NOT transposed. The flag is set when the input image is transposed before applying the image understanding sequence. Therefore the output must be changed to obtain the values that correspond to applying the sequence to the input image not transposed. This entails a simple switching of values. The features obtained are symmetric with respect to a transpose so that the centroid column value becomes the centroid row value (and the centroid row value becomes the centroid column value). When the centroid column and row values are switched, they represent the features that would have been derived from the image if the image had NOT been transposed. The GLCM descriptors have a similar pairing because of the relationship between the directions chosen. For these features, if the transpose has occurred, the 0 degree and the 90 degree descriptor values are switched. The GLCM descriptor values for 45 degrees and 135 degrees are unaffected by the transpose. A 45 degree vector is along the axis of the transpose and remains 45 degrees. A 135 degree vector transposed becomes a 315 degree vector, which is along the same ray as 135, but in the opposite direction. Since the GLCM descriptors are being calculated using sum and difference histograms that are equivalent for directions 180 degrees apart, no change is required for the 135 degree descriptors. Once these swaps are complete, the full set of features represents values for an image that is not transposed. The other features are not affected by the transposition of the input image.
Calls:	deleteFirstFeature() errorMessage()
(system)	fclose() fopen ()
Return:	WRITE_OUTPUT_SUCCESS WRITE_OUTPUT_OPEN_FILE_ERROR WRITE_OUTPUT_WRITE_FILE_ERROR WRITE_OUTPUT_CLOSE_FILE_ERROR

```
{ /* begin writeOutput */
   open file (or stdout if no input filename given)
```

```
   if error
      call errorMessage("writeOutput: error opening output file\n")
      delete all items on features
      return WRITE_OUTPUT_OPEN_FILE_ERROR

   /*
    * Write out the features for each object on the features list.
    * If the transposeFlag has been set, "transpose" the features
    * (switch centroid column and row values, and switch 0 and 90 degree
    * descriptor values) before writing out their values.  Delete each
    * featureEntry structure after it is processed.
    */
   Loop for each item on features {
      if (transposeFlag == TRUE) {
         switch centroid column and centroid row values
         switch descriptor values for  0 and  90 degrees
      } /* end if transposeFlag */
      write out features
      if error writing out a feature
         call errorMessage("writeOutput: error writing output\n")
         delete all items on features
         return WRITE_OUTPUT_WRITE_FILE_ERROR
      call deleteFirstFeature(item)
      item = nextItem
   } /* end loop for each item */

   close output file
   if error closing output file
      call errorMessage("writeOutput: error closing output file\n")
      return WRITE_OUTPUT_CLOSE_FILE_ERROR

   Return WRITE_OUTPUT_SUCCESS
} /* end writeOutput */
```

B.9 FILTERING MODULE

B.9.1 filterImage

Name:	filterImage
Input:	*K* kernel, *V* image
Output:	*W* image
Description:	This routine calculates *W* image, where *W = V – [(V eroded K) dilated K]*. Note that the input image *V* is READONLY and an output buffer is given for image *W*. The function is implemented to minimize memory required (to be able to handle very large input images). Assumptions: 1) images *V* and *W* have the same dimension 2) kernel dimension is smaller than image *V* dimensions 3) kernel dimensions are odd 4) kernel has at least one non-zero pixel (defining shape) First erode image, *W = V eroded K*, with a call to *erodeImage*, and then compute the dilation and subtraction simultaneously in-place, with a call to *dilateAndSubtract*. Then zero out the shell ((kernelNumRows-1) top and bottom,

	(kernelNumColumns-1) right and left) were there isn't enough valid data to compute valid output data for a sequence of two morphological operations (as stated in the specification [AAEC-1]). An error will be returned if there is an error received by *dilateAndSubtract* from a memory allocation failure.
Calls:	erodeImage()
	dilateAndSubtract()
(system)	malloc() free()
Return:	FILTER_IMAGE_SUCCESS FILTER_IMAGE_ALLOC_ERROR

```
{ /* begin filterImage */

   /*
    * Assumptions:
    *    1) images V and W have the same dimension
    *    2) kernel dimension is smaller than image V dimensions
    *    3) kernel dimensions are odd
    *    4) kernel has at least one non-zero pixel (defining shape)
    */
   assert(dimensions of V same as dimensions of W)
   assert(kernel is smaller than image)
   assert(kernel has dimensions that are odd)
   assert(kernel has at least one pixel > 0)

   /*
    * This implementation is designed to minimize memory required, to
    * perform calculations in-place (when no buffers are available for
    * use), and to be fast.  Handle the outer shell, where there is not
    * enough input data to generate valid output data (see
    * specification), at the end of routine (set the pixels
    * in this area to zero).
    *
    * Note that for the image understanding sequence, this routine is
    * guaranteed to have a read only input and an output buffer.  (The
    * image V is needed within the components that follow.) Therefore,
    * the function can NOT be totally in-place.  Perform the first
    * operation (the erode) using the output buffer.  Then calculate
    * the dilate and the subtraction in-place in the output buffer.
    */

   /*
    * Perform the erosion first.
    */
   call erodeImage(kernel, V image, W image)

   /*
    * Next is the dilation and subtraction.
    */
   returnCode = dilateAndSubtract(kernel, V image, W image)
   if (returnCode == DILATE_AND_SUBTRACT_ALLOC_ERROR)
      call errorMessage("filterImage: ")
      return FILTER_IMAGE_ALLOC_ERROR
```

```
/*
 * Zero out outer row edges (portions where there is not enough
 * valid input data to compute valid output data.  (Contiguous data
 * access as single loop incrementing image pointer.)  This shell is
 * twice the size for a single morph operation because this module
 * computes two sequential morphological operations.
 */

/* firstRow - first row index processed */
/* lastRow  - last row index processed */
firstRow = 2 * halfKernelNumRows
lastRow  = (filtered numRows) - (2 * halfKernelNumRows) - 1
firstColumn = 2 * halfKernelNumColumns
lastColumn  = (filtered numColumns) - (2 * halfKernelNumColumns) - 1
for row = firstRow to lastRow
   /* zero out left columns */
   for column = 0 to firstColumn - 1
      output = 0
   end for /* column */
   /* zero out right columns */
   for column = lastColumn + 1 to (detected numColumns - 1)
      output = 0
   end for /* column */
end for /* row */

/* zero out top rows (including left & right column area */
dataPointer = &w(0,0)
numElmsBorder = (kernel numRows - 1) * (filtered numColumns)
for index = 0, numElmsBorder
   *dataPointer = 0
   dataPointer += 1
end for /* index */

/* zero out bottom rows (including left & right column area */
dataPointer = w(input numRows - kernel numRows - 1,0)
for index = 0, numElmsBorder
   *dataPointer = 0
   dataPointer += 1
end for /* index */

return(FILTER_IMAGE_SUCCESS)
} /* end filterImage */
```

B.9.2 erodeImage

Name:	erodeImage
Input:	*K* kernel, *V* image
Output:	*W* image
Description:	This routine calculates *W* image, where *W* = *V eroded K*. Note that the input image *V* is READONLY. Assumptions: 1) images *V* and *W* have the same dimension 2) kernel dimension is smaller than image *V* dimensions 3) kernel dimensions are odd

4) kernel has at least one non-zero pixel (defining shape)

For morphological calculations, any kernel pixel or image pixel that has a zero value requires NO OPeration at all. Since the input image contains short integer values, the probability of it containing a value equal to zero is small. The *kernel,* however, is of type unsigned char and may very well have some zero values. To take advantage of these NOOPs and increase processing speed, loop over the kernel first and then over the image while calculating the erosion. Then, for any kernel pixel that is zero, the input image does not need to be referenced at all and processing can jump to the next kernel pixel! Note that the erosion for any pixel is not fully computed until the outer loop over the entire kernel completes.

The shell around the image ((kernelNumRows-1)/2 rows of pixels on the top and bottom of the image, and (kernelNumColumns-1)/2 columns of pixels to the left and right of the image) is set to zero because not enough valid data exists to calculate valid output data in this shell.

Calls: none

Return: none

```
{ /* begin erodeImage */

   /*
    * Assumptions:
    *    1) images V and W have the same dimension
    *    2) kernel dimension is smaller than image V dimensions
    *    3) kernel dimensions are odd
    *    4) kernel has at least one non-zero pixel (defining shape)
    */
   assert(dimensions of V same as dimensions of W)
   assert(kernel is smaller than image)
   assert(kernel has dimensions that are odd)
   assert(kernel has at least one pixel > 0)

   /* initial output buffer to a value (SHORT_INT_MAX) favoring the
    * data - performing a MIN will favor the data
    */
   loop for each pixel w(x,y)
      w(x,y) = SHORT_INT_MAX
   end /* loop over w(x,y)

   /*
    * Perform the erosion.
    * For morphological calculations, any kernel pixel or image pixel
    * that has a zero value requires NO OPeration at all.  Since the
    * input image contains short integer values, the probability of it
    * containing a value equal to zero is small.  The kernel, however,
    * is of type unsigned char and may very well have some zero values.
    * To take advantage of these NOOPs and increase processing speed,
    * loop over the kernel first and then over the image while
    * calculating the erosion.  Then, for any kernel pixel that is zero,
    * the input image does not need to be referenced at all and
    * processing can jump to the next kernel pixel! Note that the
    * erosion for any pixel is not fully computed until the outer loop
    * over the entire kernel completes.
```

```
 */
loop for each pixel k(m,n)
   if k(m,n) > 0
      loop for each pixel v(x,y) where (M-1)/2<=x<X-(M-1)/2
            and (N-1)/2<=y<Y-(N-1)/2
         w(x,y) = MIN(w(x,y), v(x+m,y+n))
      end /* loop over v(x,y) */
   end /* if k(m,n) > 0 */
end /* loop over k(m,n) */

/*
 * Zero out the unprocessed shell where there is not enough valid
 * data to generate valid output data.
 */
loop over top rows, bottom rows, left columns, and right columns of
   W in the unprocessed shell
   w(x,y) = 0
end /* loop over shell */

return

} /* end erodeImage */
```

B.9.3 dilateAndSubtract

Name:	dilateAndSubtract
Input:	*K* kernel, *V* image, *W* image
Output:	*W* image
Description:	This routine calculates *W* image, where *W = V – [W dilated K]*. Note that the input image *V* is READONLY. The function is implemented to minimize memory required (to be able to handle very large input images). Assumptions: 1) images *V* and *W* have the same dimension 2) kernel dimension is smaller than image *V* dimensions 3) kernel dimensions are odd 4) kernel has at least one non-zero pixel (defining shape) 5) The shell ((kernelNumRows-1)/2 top and bottom, (kernelNumColumns-1)/2 right and left) is set to zero in the calling routine (NOT here). Dilate and subtraction simultaneously in-place. This implementation is optimized to minimize memory required, to perform calculations in-place (the input is overwritten and becomes the output), and to be fast. Use a temporary buffer to store 'half the number of rows of a kernel' amount of rows of the image *W* allowing an in-place calculation to be performed.
Calls: (system)	free() malloc() memcpy()
Return:	DILATE_AND_SUBTRACT_SUCCESS DILATE_AND_SUBTRACT_ALLOC_ERROR

```
{ /* begin dilateAndSubtract */
   /*
    * Assumptions:
    *     1) images V and W have the same dimension
    *     2) kernel dimension is smaller than image V dimensions
    *     3) kernel dimensions are odd
    *     4) kernel has at least one non-zero pixel (defining shape)
    *     5) The shell ((kernelNumRows-1)/2 top and bottom,
    *         (kernelNumColumns-1)/2 right and left) is set to zero in
    *         the calling routine (NOT here)
    *
    */
   /* assert assumptions about images */
   assert(dimensions of V same as dimensions of W)
   assert(kernel is smaller than image)
   assert(kernel has dimensions that are odd)
   assert(kernel has at least one pixel > 0)

   alloc tempEroded space [(kernel numRows)/2 - 1] x input numColumns
   if alloc fails
      call errorMessage("dilateAndSubtract: error allocating buffer\n")
      return DILATE_AND_SUBTRACT_ALLOC_ERROR

   /*
    * Perform the dilation and subtract in place using tempEroded.
    * Can't use the same trick as for the erode to increase speed
    * because from here on, calculations must be in place.  The buffer
    * is used to store the first top half of a kernel's worth of image
    * pixels since the in-place calculations will overwrite all pixels
    * above and to the left of the current pixel (when the image is
    * processed top row to bottom row, and from left column to right
    * column.  After each row is processed, the buffer must be cycled
    * to represent the correct rows associated to the new current row.
    */
   halfKernelNumRows = (kernel numRows - 1)/2
   halfKernelNumColumns = (kernel numColumns - 1)/2
   /* firstColumn - first column index processed */
   /* lastColumn  - last column index processed */
   firstColumn = 2 * halfKernelNumColumns
   lastColumn  = (filtered numColumns) - (2 * halfKernelNumColumns) - 1
   /* firstRow - first row index processed */
   /* lastRow  - last row index processed */
   firstRow = 2 * halfKernelNumRows
   lastRow  = (filtered numRows) - (2 * halfKernelNumRows) - 1

   initialize tempEroded
   for row = firstRow to lastRow
      update tempEroded
      for column = firstColumn to lastColumn
         Set up some bookkeeping so that the loop that follows knows
            where to access the data
            (from tempEroded or from the input buffer)
         /* calculate dilate pixel */
         loop for each pixel k(m,n)
            maxval = MAX(maxval, w buffer (from tempEroded or buffer))
            update tempEroded as needed
         end /* loop over k(m,n) */
```

```
            calculate filtered value (subtract dilate pixel from original)
            w(column, row) = v(column, row) - w(column, row)
            store output in place
        end for /* column */
    end for /* row */

    /* free temporary buffers required to compute in-place */
    free tempEroded
    return(DILATE_AND_SUBTRACT_SUCCESS)
} /* end dilateAndSubtract */
```

B.10 REGION SELECTION MODULE

B.10.1 selectRegions

Name:	selectRegions	
Input:	*V* image	input image
	thresholdLevel	input threshold, above which a pixel is considered "target"
	minArea	minimum acceptable area criteria
	maxRatio	maximum acceptable perimeter/area ratio
	selectNumber	number of objects to retain (largest values) after ranking
	W image	filtered image
Output:	*W* image	labeled image (filtered image overwritten by labeled image)
	regions	object list containing bounding box, initial features, and
	alias	lists for each object
Comment:	This routine is the highest level routine for the Region Selection module and provides an interface with the mainline. First the image *W* is thresholded to determine which pixels are considered "on target". Then these target pixels are grouped so that contiguous pixels form regions. The value SHORT_INT_MIN is used to signify a background pixel value. As the objects are formed, features are calculated in an incremental fashion. Whenever two objects are merged, the partial features are also merged to represent the partial features for the merged object. Once all the regions have been determined, the feature computations are completed. Then a selection criteria is used to select a subset of regions of interest(ROIs) to retain. The outputs of this routine are: a labeled image, *O* and a list of selected ROIs (with feature values), *regions*. The labeled image consists of the pixel value SHORT_INT_MIN for every background pixel, and an index/label value for every target pixel. These indices can be used with the information from *regions* to determine the location and shape of an ROI. To minimize memory requirements, *selectRegions* uses the input filtered image *W* to store the values in the labeled image *O*. Many algorithms exist to group neighboring pixels into a connected region. Two types of these connected component algorithms were considered for implementation. One uses the seed fill approach by [Heckbert] and the other is a raster scan approach described by [Lumia]. The seed fill algorithm takes a seed point and connects neighbors with the same value as the seed point to a given new	

value. A stack is used to store neighboring pixels of interest and processing continues until the stack is empty. This algorithm is often useful in paint utilities and was not used here because of the irregular memory access required and the possibility of the stack growing unmanageable. However, this algorithm, as well as others that exist, should be considered as a viable alternative for others who implement this benchmark.

The raster scan approach described in [Lumia] contains three variations. The first variation traverses the image line by line assigning pixels to objects and updating an equivalence table that keeps track of the objects that have been merged. As the image is traversed and distinct labels are assigned to pixels to represent distinct objects, it is possible for two separate objects to become connected. Then these two connected objects are really one object and need to be merged into one object. This is achieved by associating multiple labels to one object, where one of these labels will be defined as the label for that object and the other labels will be aliases for that object. Therefore, when the end of the image is processed, each distinct object is labeled and may have any number of aliases associated to it. These aliases are stored on the equivalence table defining which labels go to which object. Thus, for large images with complex objects, the size of the equivalence table may grow to an unmanageable size. The second variation has no equivalence table and iterates on the image in two passes: one from top to bottom and the other from bottom to top. While the image is traversed a flag is set if any objects were merged. The iterations continue until an iteration occurs that does not have the flag set (where no objects were merged). This variation requires an undetermined number of passes through the image, which may be very large. The third variation is a hybrid of the other two where the image is traversed twice: once top to bottom and then from bottom to top. For each image line, an equivalence table is kept to keep track of merged objects (which is used to relabel the objects). While each image line is processed, an equivalence table keeps track of merged objects. At the end of the image line, if any objects have been merged, the merged pixels are relabeled using the data on the equivalence table. This method has a smaller equivalence table and a predetermined number of passes through the image, but may require revisiting many pixels in order to relabel objects.

For this benchmark, assuming that the targets (or objects) to be labeled are never going to be very large compared to image size or very intricate in terms of shape, the first raster scan approach has been implemented. The size of the equivalence table, however, is bounded and a relabeling procedure is performed whenever the limits of the table are reached.

The algorithm used, for the baseline implementation, to group neighboring pixels into regions is described as follows. As the image *W* is traversed, any pixel value larger than *thresholdLevel* is considered a target pixel. The image is traversed from the top row to the bottom row and from the left column to the right column. The neighbors of a target pixel (above and to the left – or neighboring pixels that have already been processed) are checked to see if any are labeled. If none of the neighbors are labeled, then the target pixel represents a new object and a new object label is used. If only one neighbor is labeled, then label the target pixel with the same label. If more than one neighbor has been labeled: if they are labeled with the same label, label the target pixel with that label; if they are labeled with different labels, then use the label from one neighbor while insuring that an alias equal to the labels from the other neighbors exist in the table. The image is traversed once, processing each pixel one at a time until all the pixels have been processed and assigned a label.

This implementation does not allow an object pixel to be on the outer shell (one pixel in width) of the image. Furthermore, the pixels in this outer shell are disqualified as target pixels and are explicitly set to the background value. Since the filtering module, which precedes the region definition, contains a two-step kernel process, unless the kernel is one pixel wide, it is guaranteed that the image will have a border of undefined data (set to zero) greater than one pixel. Thus, when the image is traversed to group neighboring pixels, a faster single looping construct can be used rather than the slower traditional double looping construct (one for rows and one for columns). And since the outer shell will never have a target pixel, when neighboring pixels of a target pixel (not on the outer shell) are accessed, special cases at the boundary are avoided.

The initial features are not dependent on spatial relationships between pixels within the same object (or ROI) and can be calculated incrementally. Thus, as each target pixel is visited and assigned a label, its contribution to the features is calculated and stored on the **ObjectEntry** structure. The output of this module, *regions*, is list of objects or a linked list of **ObjectEntry** structures. This linked list of objects may contain a linked list of aliases for each object using the **AliasEntry** structure. The label value associated to the object, a bounding box around the object, initial features for the object, and a ranking metric (used in to rank the objects) completes the members on an **ObjectEntry** structure. The structure used to store the initial features, **SomeFeatures**, is used to store the incremental values composing the feature until the entire object is defined. These incremental features are changed (consisting of a normalization where appropriate) to represent the features defined in the specification – *centroid, area, perimeter, mean,* and *variance.*

If the features were not calculated incrementally, then the ROIs must be traversed after grouping pixels into ROIs to calculate the features. This would require multiple passes of the image, once to segment the pixels, and once to calculate the features. Thus, this incremental calculation eliminates the need for multiple passes and increases the speed of the baseline implementation.

An object entry link is formed when more than one object is found in the image where each entry on the list is a distinct object in the image. Whenever two objects are found to be connected, they are merged (associated together where one

	represents an alias of the other). This is achieved by deleting one of the merging objects from the object list and creating an alias with the same label as the deleted object onto the alias list for the other merging object. The list *regions* is a linked list of linked lists. One type of link is an object link and the other is an alias link. While the image is being traversed and the ROIs are being connected, many objects may be merged together. Each time a merge takes place, an entry is deleted from the object list and an alias with the same label is added to different alias list keeping track of these connections. The bounding box definition and the initial feature calculations for each object are also combined before the object is deleted from the object list and the alias is placed on the alias list. This entails consolidating information on the **ObjectEntry** structures or merging the statistics of the two objects to create statistics representative of the merged object. Once all the pixels in the image have been assigned to an ROI, then a final computation step is performed. This consolidation step is required to normalize the *centroid, mean,* and *variance* since it is now possible to calculate the total number of pixels in an ROI. The *area* feature does not require the normalization step. This consolidation step is also used to prune out objects that do not satisfy the criteria driven by *minArea* and *maxRatio*. At the end of this module, *regions* contains a list of objects where each entry on the list represents an ROI that passes the *minArea* and *maxRatio* criteria. The members of the **ObjectEntry** structure contain for each ROI: 1) the labels used in image *O* to label the ROI (one label plus any number of alias labels contained in an alias linked list) 2) the bounding box around the ROI 3) initial features *centroid, area, perimeter, mean,* and *variance,* and 4) the ranking metric (*mean*area*) to be used when the ranking selection criteria is applied. The list *regions* is a list of lists - a list of the different regions, and a list of aliases for a particular region. Assumptions: 1) 8-neighbor definition is used 2) *V* and *W* images are all type ShortInt, and have the same dimensions 3) *minArea* > 0 4) *maxRatio* > 0 5) *selectNumber* > 0 6) *minArea* < total area of input image V 7) *selectNumber* $< 2^{16} - 2$
Calls: (system)	addObject(), computeFeatures(), findConnection(), getLabel(), selectSubset(), updateObject(),
Return:	SELECT_REGIONS_SUCCESS SELECT_REGIONS_ALLOC_ERROR

```
#define  MAX_NUM_LABELS 2^16 - 2   /* max num of labels possible */

{ /* begin selectRegions */
```

```
/*
 * This function assumes that the outer shell of the image can not
 * contain a target pixel.  Furthermore, this function explicitly
 * disqualifies the pixels in the outer shell from being considered
 * as a target pixel. (The outer shell consists of the pixels on
 * the top row, bottom row, left most column, and right most column.)
 * This assumption is made to increase the speed of image access by
 * using a single loop over the image rather than two embedded loops.
 * Since the left and upper neighbors of each target pixel are
 * also accessed, there would be boundary cases introduced for
 * any target pixel that is on the outer shell. However, this design
 * does NOT change the results of the image understanding sequence
 * because this stage is preceded by the filtering component unless
 * the kernel has a dimension of one. This filtering component has a
 * two-stage kernel operation which requires the outer shell to be
 * zeroed out with the size of the shell equal to the dimension of
 * the kernel minus one. If the kernel dimension is larger than one,
 * then the outer shell required will be at least one pixel wide.
 *
 * This function loops through all the pixels in the image W and tags
 * all pixels > thresholdLevel as "on target" pixels.  These target
 * pixels are grouped so that contiguous adjacent form a single
 * object.  For each target pixel determine the label to use for that
 * pixel.  If the pixel is isolated, then it represents a new object.
 * If neighbors of the pixel are labeled, then the pixel belongs to
 * the same object as its neighbor.  If more than one neighbor is
 * labeled with different labels, then objects must be merged
 * together into a single object (through an alias list).  Thus, for
 * each target pixel, the neighbors above and to the left of that
 * target pixel are checked to determine whether any neighbors have
 * already been labeled.  The neighbors that are checked are
 * depicted to the right where a '?' denotes
 * a neighbor that is checked, an 'x' denotes        ? ? ?
 * the target pixel being labeled, and a '-'         ? x -
 * denotes a neighbor that is NOT checked.           - - -
 *
 * Any pixel not in the outer shell has all of these neighbors, and
 * is not a boundary case.  For example, all the pixels in the top
 * row are boundary cases and have no neighbors above them.  Since
 * the outer shell is disqualified, none of these pixels will pass
 * as a target pixel.  Hence a pixel in the outer shell (the
 * boundary cases) will never go through the neighborhood test.
 * Therefore, since it is guaranteed that there will be no boundary
 * cases, the implementation can take advantage of this and be
 * written efficiently and concisely (ignoring all boundary cases).
 *
 * The neighbors of a target pixel are checked to determine whether
 * the target pixel already belongs to an object, or is a new object.
 * If more than one of the neighboring pixels are labeled, then a
 * neighbor's label is used.  When objects are merged together, the
 * object and alias lists are updated to reflect this merge.
 *
 * For faster pixel access, since the 2-dimensional image is stored
 * in row dominant order, an optimal single loop pixel pointer is
 * used to access the image from the left column to the right
 * column and from the top row to the bottom row.
 *
```

```
 * Assumptions:
 * 1) 8-neighbor definition is used
 * 2) V and W images are all type ShortInt, and have the same
 *    dimensions
 * 3) minArea > 0
 * 4) maxRatio > 0
 * 5) selectNumber > 0
 * 6) minArea < total area of input image V
 * 7) selectNumber  < 2^16 - 2
 *
 * Set backgroundValue = SHORTINT_MIN, with labels starting at
 * backgroundValue+1
 *
 * The ObjectEntry structure is used for a linked list of linked
 * lists where the nextObject points to the next object in the list
 * and the aliasList points to an alias list for the current object.
 */
assert(V and W are all ShortInt with same dimensions)
assert(minArea > 0)
assert(maxRatio > 0)
assert(selectNumber > 0)
assert(minArea < total area of input image V)
assert(selectNumber < 2^16 - 2)

/*
 * Set outer shell (one pixel wide) to nonTargetValue, disqualifying
 * all pixels in outer shell from being a target value (which they
 * will never be since this process is preceded by the filtering
 * component - see comment above).
 */
backgroundValue = SHORTINT_MIN
nonTargetValue = backgroundValue
loop over top row, bottom row, column=0, column=numColumns-1
   W(pixel) = nonTargetValue
endloop

/*
 * The connected component algorithm used is described above.
 * See the first algorithm described by R. Lumia, L. Shapiro, and
 * O. Zuniga in "A New Connected Components Algorithm for Virtual
 * Memory Computers," Computer Vision, Graphics, and Image
 * Processing  22, pp. 287-300, 1983.
 *
 * Traverse the filtered image line by line once, thresholding to
 * distinguish target pixels from background pixels.  Once a target
 * pixel is found, determine which label to assign it by looking at
 * neighbors above and to the left of the current target pixel.  If
 * no neighbors are labeled, then the target pixel starts a new
 * object.  If only one neighbor is labeled, use that label.  If
 * more than one neighbor is labeled (with different labels), then
 * objects must be merged into one object.  When two objects merge,
 * keep track of this merge through the an alias list.
 * The list, regions, contains a list of all regions; along with a
 * list of the aliases associated to each region (if any).  The list
 * is be used later to traverse through all the objects in the image
 * and finish the feature calculation for each object, and to rank
 * and cull the regions.
```

```
 */
regions     = NULL; /* an empty list to start with */
imageStart   = &W(1,1)
imageEnd     = &W(numColumns-1,numRows-2)  /* don't do last row */

/*
 * It is faster to traverse the image with a single pointer using a
 * single loop rather than two nested loops, but the centroid feature
 * requires the row and column index.  The variable pixelLocation is
 * used to derive the row and column index from a single pixel index.
 */
pixelLocation = numColumns

/* offsets for neighbors of the labeled image */
offsetTopLeftNeighbor  = (-numColumns-1)
offsetTopNeighbor      = (-numColumns)
offsetTopRightNeighbor = (-numColumns+1)
offsetLeftNeighbor     = -1

ShortInt neighbor(4), newLabel
loop for imagePtr=imageStart, imagePtr <= imageEnd, imagePtr+=1 {
   pixelLocation += 1;
   if (*imagePtr > thresholdLevel) {    /* pixel is "on target" */
      /* Any neighbors (above or to left) of target pixel labeled? */
      neighbor(0) = *(labelPtr+offsetLeftNeighbor)     /* 1 2 3 */
      neighbor(1) = *(labelPtr+offsetTopLeftNeighbor) /* 0 x - */
      neighbor(2) = *(labelPtr+offsetTopNeighbor)     /* - - - */
      neighbor(3) = *(labelPtr+offsetTopRightNeighbor)
      if ((neighbor(0) == backgroundValue) &&
          (neighbor(1) == backgroundValue) &&
          (neighbor(2) == backgroundValue) &&
          (neighbor(3) == backgroundValue)) { /* new object */
         newLabel    = getLabel(pixelLocation, O, &regions)
         if (newLabel != backgroundValue) { /* valid label */
            returnCode = addObject(newLabel,&regions,&currentObject)
            if (returnCode == ADD_OBJECT_ALLOC_ERROR)
               call errorMessage("selectRegions: ")
               return SELECT_REGIONS_ALLOC_ERROR
            }  /* end if addObject error */

            /* calculate partial feature values */
            call updateObject(backgroundValue, W, O, pixelLocation,
                              currentObject)
         } /* end valid label */
      } /* end if new object */
      else { /* one or more neighbors labeled */
         returnCode = findConnection(backgroundValue, neighbor,
                             &regions, &newLabel, &currentObject);
         if (returnCode != FIND_CONNECTION_ALLOC_ERROR)
            call errorMessage("selectRegions: ")
            return SELECT_REGIONS_ALLOC_ERROR

         /* calculate partial feature values */
         call updateObject(backgroundValue, W, O, pixelLocation,
                           currentObject)
      }  /* end at least one neighbor labeled */
```

```
            /* add label to object image */
            *labelPtr = newLabel;

        } /* end if label is targetValue */
        else {      /* add backgroundValue to object image */
            *labelPtr = backgroundValue;
        } /* end else label is backgroundValue */
    } /* end loop for each pixel */

    /*
     * Finish calculating features and select ROIs.
     */
    numberObjects = computeFeatures(minArea, maxRatio, &regions)

    /*
     * Apply the ranking selection criteria to the selected ROIs.
     */
    call selectSubset(numberObjects, selectNumber, &regions)

    return SELECT_REGIONS_SUCCESS

} /* end selectRegions */
```

B.10.2 addObject

Name:	addObject	
Input:	*newLabel*	label to assign to new object
	regions	pointer to linked list of objects to add new object to
Output:	*regions*	pointer to linked list of objects with new object added
	newObject	pointer to new object added
Comment:	This routine adds an objectEntry structure to the given linked list after creating the new structure. The new object added to the list is prepended to the linked list and is assigned the labelValue, *newLabel*. The new head of the revised list *regions* and the *newObject* are both returned. This routine does not check to see if an object with value newLabel already exists on the list. (Therefore if it is called twice with the same label, two objects will be placed on the list with identical label values.)	
Calls:	createObject()	
Return:	ADD_OBJECT_SUCCESS ADD_OBJECT_ALLOC_ERROR	

```
{ /* begin addObject */

    /*
     * Limitation: This routine does not check to see if an
     * object with the newLabel already exists on the list.
     * Create a new object, updating pointer to the new object.
     */
    object = createObject();
    if (! object)
        call errorMessage("addObject: ")
        return(ADD_OBJECT_ALLOC_ERROR);
```

```
   /*
    * Assign the label to the new object, and add new object
    * to the beginning of linked list.  Update pointer to list.
    */
   object->labelValue = newlabel
   object->nextObject = *regions
   if (*regions != NULL)
      *regions->lastObject = object

   *regions   = object
   *newObject = object

   return(ADD_OBJECT_SUCCESS);
} /* end addObject */
```

B.10.3 computeFeatures

Name:	computeFeatures	
Input:	*minArea*	minimum acceptable area value for an object
	maxRatio	maximum acceptable perimeter/area ratio for an object
	regions	list of objects to finalize initial feature calculations and apply the acceptance criteria defined by *minArea* and *maxRatio* to.
Output:	*regions*	with objects initial features finalized and acceptance criteria satisfied
Comment:	This routine takes for input a linked list of objects. Features associated to each object are finalized (the *centroid, mean,* and *variance* are normalized since it is now possible to calculate the total number of pixels in an object). The *area* feature does not require the normalization step. As the features are calculated, the values are screened to see if they pass the selection criteria set by *minArea* and *maxRatio*. Any object that does not satisfy the *minArea* and *maxRatio* is removed from the object list. The ranking metric (*mean*area*) is also calculated for each object that passes to be used later on when the ranking criteria is applied. The number of objects on the linked list of objects, *regions,* is returned.	
Calls:	deleteObject()	
Return:	numberObjects	

```
{ /* begin computeFeatures */

   /*
    * Finish feature calculation, perform normalization as required.
    */
   numObjects = 0;
   currentObject = *regions;
   while (currentObject != NULL) { /* loop for each object */
      nextObject = currentObject->nextObject;

      /*
       * Normalize the features that require normalization that
       * are used in the selection criteria.
       */
```

```
      area = currentObject->initFeatures.area;
      currentObject->initFeatures.mean =
            currentObject->initFeatures.mean / area;

      /*
       * Apply part of the selection logic.  If the object passes the
       * test, compute the rest of the initial features.  Otherwise,
       * delete the object from the list.
       */
      if ((minArea <= currentObject->initFeatures.area) AND
          ((currentObject->initFeatures.perimeter /
            currentObject->initFeatures.area) <= maxRatio) ) then {
         numObjects += 1;

         /* Finish calculating rest of initial features */
         currentObject->initFeatures.centroidRow =
               currentObject->initFeatures.centroidRow / area;
         currentObject->initFeatures.centroidColumn =
               currentObject->initFeatures.centroidColumn / area;
         mean = currentObject->initFeatures.mean;
         currentObject->initFeatures.variance =
               (currentObject->initFeatures.variance / area) -
               (mean * mean);

         /* Calculate ranking metric (mean*area) */
         currentObject->rankMetric =
               currentObject->initFeatures.mean * area;
      } else { /* delete object that doesn't pass criteria */
         deleteObject(currentObject, regions);
      }
      currentObject = nextObject;
   } /* end while currentObject */

   return numObjects
} /* end computeFeatures */
```

B.10.4 createAlias

Name:	createAlias
Input:	none
Comment:	This routine allocates memory for an aliasEntry structure and initializes all members to zero or null (for pointers). These aliasEntry structures are used to compose alias lists for an object.
Calls:	
(system)	malloc()
Return:	pointer to aliasEntry

```
{ /* begin createAlias */
   allocate memory for aliasEntry structure
   if alloc failed
      call errorMessage("createAlias: error allocating memory\n")
      return(NULL)
   initialize members in aliasEntry structure
      (pointers to null, label value to zero)
```

```
   return(newAlias)
} /* end createAlias */
```

B.10.5 createObject

Name:	createObject
Input:	none
Comment:	This routine allocates memory for an objectEntry structure and initializes all members to zero or null (for pointers). The objectEntry structure is used to keep track of each object or ROI found. There are members on the structure to: 1) contain a bounding box definition around the object, 2) retain a label value associated to the object, 3) maintain an alias list for the object if more than one label is used for the object, and 4) keep incremental features (which become final feature calculations later).
Calls:	
(system)	malloc()
Return:	pointer to objectEntry

```
{ /* begin createObject */
   allocate memory for objectEntry structure
   if alloc failed
      call errorMessage("createObject: error allocating memory\n")
      return(NULL)
   initialize members in objectEntry structure
      (pointers to null, initialize feature values to zero)
   initialize bounding box values
      newObject->box.upperLeft.column    = INT_MAX
      newObject->box.upperLeft.row       = INT_MAX
      newObject->box.lowerRight.column   = -1
      newObject->box.lowerRight.row      = -1
   return(newObject)
} /* end createObject */
```

B.10.6 deleteAlias

Name:	deleteAlias
Input:	*aliasEntry* pointer to aliasEntry
Comment:	This routine frees memory associated to the *aliasEntry* structure. If a NULL pointer is given, then the routine simply returns. Otherwise, the pointer is assumed to point to an AliasEntry structure whose memory is freed.
Calls:	
(system)	free()
Return:	none

```
{ /* begin deleteAlias */
   while (aliasEntry != NULL)
      nextAlias = aliasEntry->nextAlias
      free aliasEntry structure
      aliasEntry = nextAlias
   end /* while */
```

```
   return
} /* end deleteAlias */
```

B.10.7 deleteObject

Name:	deleteObject	
Input:	*currentObject*	pointer to ObjectEntry to delete
	listHandle	pointer to linked list containing *currentObject*
Output:	*listHandle*	pointer to linked list with *currentObject* deleted
Comment:	This routine removes *currentObject* from the linked list and then frees memory associated to the ObjectEntry structure. If a NULL pointer is given, then the routine simply returns. Otherwise, the pointer is assumed to point to an ObjectEntry structure whose memory is freed. The pointer to the linked list is updated and returned (*currentObject* may be the first element on the list).	
Calls:	deleteAlias()	
	removeObject()	
(system)	free()	
Return:	none	

```
{ /* begin deleteObject */
   if (currentObject != NULL) {

      newList = *listHandle

      /*
       * Reset listHandle if object is the first element on the list.
       */
      if (currentObject == *listHandle)
         newList = currentObject->nextObject;

      /*
       * Remove object from list then delete structure.
       */
      removeObject(currentObject, listHandle)

      /*
       * Delete structures within the object entry structure first.
       */
      deleteAlias(currentObject->aliasList);

      free objectEntry structure

      *listHandle = newList
   } /* end non-NULL pointer given */

   return
} /* end deleteObject */
```

B.10.8 findConnection

Name:	findConnection

Input: backgroundValue value used to denote a background pixel
neighbor 4 element array with label values for neighbors n0-n3
regions linked list of objects found so far

Output: *regions* linked list of objects found so far (modified if two objects are merged)
newLabel label that the pixel should be given
currentObject object associated to the *newLabel*

Comment: This routine finds the connections between a pixel and its neighbors to determine the label and object to assign to the pixel. If more than one neighbor is labeled with different values, then the objects are merged (one object becomes an alias for the other object.

Assumptions: 1) at least one neighbor is labeled,
2) the image is being traversed from top row to bottom row and from left column to right column, and
3) the neighbor index scheme as shown below where x is the pixel being labeled and 0-3 are indices for the 4 element array and for the neighbors of interest and '-' denotes a neighbor in any state (not of interest).

```
n1 n2 n3
n0  x  -
 -  -  -
```

With the assumptions above, minimize the operations (including the number of neighbors to check), with the following algorithm: Check n0 first.

Case 1: n0 labeled
Set *newLabel* equal to n0's label. Check n3. If n3 labeled (and not equal to n0) merge objects for n0 and n3. Finished
Note that if n0 is labeled, there is no need to check n1 or n2. If either one is labeled, they would have the same label as n0. Need to check n3 in case of a merge.

Case 2: n0 not labeled, check n2 next.
n2 labeled
Set *newLabel* equal to n2's label. Finished
Note that if n0 is not labeled and n2 is labeled, there is no need to check n1 or n3. If either one is labeled, they would have the same label as n2.

Case 3: n0 and n2 not labeled, check n1 next.
n1 labeled
Set *newLabel* equal to n1's label. Check n3. If n3 labeled (and not equal to n1) merge objects for n1 and n3. Finished

Case 4: n0, n1, and n2 not labeled, check n3 next.
n3 labeled
Set *newLabel* equal to n3's label. Finished

Case 5: n0, n1, n2, and n3 not labeled – should never occur – assumption is at least one of n0, n1, n2, or n3 is labeled (this special case is handled in *selectRegions*)

Calls: findObject()
mergeObject()

Return:	FIND_CONNECTION_SUCCESS FIND_CONNECTION_ALLOC_ERROR

```
{ /* begin findConnection */

   /*
    * Assumptions:
    * 1) at least one neighbor is labeled,
    * 2) the image is being traversed from top row to bottom row
*    and from left column to right column, and
    * 3) the neighbor index scheme as shown below where x is the
*     pixel being labeled and 0-3 are indices for the 4 element          *
array and for the neighbors of interest and '-' denotes a          *
neighbor in any state (not of interest).
    *        n1 n2 n3
    *        n0  x  -
    *         -  -  -
    */
   ASSERT(! (neighbor(0) == backgroundValue) AND
            (neighbor(1) == backgroundValue) AND
            (neighbor(2) == backgroundValue) AND
            (neighbor(3) == backgroundValue) )

   /*
    * If n0 labeled, then only need to check n3
    * (if n0 labeled and n1 or n2 labeled, then n0=n1=n2;
    * if n0 labeled and n1 and n2 not, then if n3 need to merge)
    */
   if neighbor(0) != backgroundValue {
      *currentObject = findObject(regions,neighbor(0))
      *newLabel = (*currentObject)->labelValue
      if neighbor(3) != backgroundValue {
         returnCode = mergeObject(neighbor(3),backgroundValue,
                                           *currentObject,regions)
         if (returnCode == MERGE_OBJECT_ALLOC_ERROR)
            call errorMessage("findConnection: ")
            return FIND_CONNECTION_ALLOC_ERROR
      } /* end n0 and n3 labeled, may need to merge */
   } /* end n0 labeled */

   /*
    * n0 not labeled, check n2 first
    * (if n2 labeled and n1 or n3 labeled, then n2=n1=n3)
    */
   elseif neighbor(2) != backgroundValue {
      *currentObject = findObject(regions,neighbor(2))
      *newLabel = (*currentObject)->labelValue
   } /* end n0 not labeled, n2 labeled no merge case */

   /*
    * n0, n2 not labeled, check n1
    * (if n1 labeled and n3 labeled, then merge)
    */
   elseif neighbor(1) != backgroundValue {
      *currentObject = findObject(regions,neighbor(1))
      *newLabel = (*currentObject)->labelValue
      if neighbor(3) != backgroundValue {
```

```
        returnCode = mergeObject(neighbor(3),backgroundValue,
                                 *currentObject,regions)
        if (returnCode == MERGE_OBJECT_ALLOC_ERROR)
           call errorMessage("findConnection: ")
           return FIND_CONNECTION_ALLOC_ERROR
      } /* end n0 and n3 labeled, may need to merge */
   } /* end n0,n2 not labeled, n1 labeled, may need to merge */

   /*
    * n0, n1, n2 not labeled, check n3, no merge case.
    */
   elseif neighbor(3) != backgroundValue {
      *currentObject = findObject(regions,neighbor(3))
      *newLabel = (*currentObject)->labelValue
   } /* end n0,n1,n2 not labeled, n3 labeled */

   /* n0, n1, n2, n3 not labeled, should never get here */
   else {
      errorMessage(
         "findConnection called with all no labeled neighbors!!\n")
   } /* end n0,n1,n2,n3 not labeled */

   return
} /* end findConnection */
```

B.10.9 findObject

Name:	findObject	
Input:	*regions*	linked list of objects to look for object
	labelValue	label associated to object to find
Output:	foundObject	(see return below)
Comment:	This routine looks on the linked list of objects for a label value. This label value may be a label associated to an object or an alias associated to an object. If the value is found the object associated to the label is returned and the object's label is returned (which is different than *labelValue* when *labelValue* is an alias of the object). If the label is not found, a NULL pointer is returned.	
Calls:		
Return:	foundObject	object on *regions* associated to *labelValue* NULL if object is not found

```
{ /* begin findObject */
   foundObject   = NULL
   currentObject = regions
   foundFlag     = FALSE

   /*
    * Find the object on linked list with the given label (or alias).
    * Return the object associated to the given label/alias.
    */
   while ((! foundFlag) && (currentObject != NULL)) {
      if (currentObject->labelValue == labelValue) { /* found label */
         foundFlag   = TRUE
         foundObject = currentObject
```

```
      }
      else { /* check aliases */
         currentAlias = currentObject->aliasList
         while ((! foundFlag) && (currentAlias != NULL)) {
            if (currentAlias->labelValue == labelValue) {
               foundFlag   = TRUE             /* found label */
               foundObject = currentObject    /* object for alias */
            }
            nextAlias    = currentAlias->nextAlias
            currentAlias = nextAlias
         } /* end while loop for aliases */
      } /* end else check aliases */
      nextObject    = currentObject->nextObject;
      currentObject = nextObject;       /* go on to next object */
   }
   return(foundObject)
} /* end findObject */
```

B.10.10 getLabel

Name:	getLabel	
Input:	pixelLocation	single index specifying location of pixel to be labeled
	O	labeled image
	regions	linked list of objects (with linked list of aliases per object)
Output:	*O*	updated if label was recycled
	regions	updated if label was recycled
Comment:	This routine gets and returns a label value to use for a new object (a target pixel NOT connected to another target pixel). An internal parameter, *currentLabel,* is kept to store the label used last and is initialized to SHORTINT_MIN – the background value. An *Int* type parameter is used for *currentLabel* so that it can contain the value SHORTINT_MAX+1. When this parameter gets too large (is larger that SHORTINT_MAX), signaling that all labels available for a *ShortInt* type value have been used, a label must be recycled. At which point, another internal parameter, *runOutLabelsFlag,* is set to signal that recycling is required. Then, a label associated to an alias if found to be recycled. The linked list *regions* is traversed until the first alias is found. This alias is recycled by removing the alias from the alias list, relabeling the object image replacing the aliasLabel with the object's labelValue. Then the label for the alias can be reused. Note that the bounding box for the object may be larger than the bounding box for just the alias. Check for the case when the *pixelLocation* is within the bounding box. If this is true, don't relabel pixels to the right of the *pixel* (they haven't been labeled yet). For this particular case, it will only be the pixels on the same row as *pixel* and to the right that have not been labeled. The image is being traversed row-by-row and column-by-column so pixels above and to the left of *pixel* have been labeled and pixels below and to the right have not been labeled. If there are no more aliases available to recycle, return the BACKGROUND_VALUE to flag this condition.	
Calls:	deleteAlias()	

(system)	floor()	
Return:	label	label to use for pixel starting a new object

```
{ /* begin getLabel */
   define MAX_NUM_LABELS = SHORTINT_MAX
   define currentLabel type static Int

   initialize currentLabel = SHORTINT_MIN
   initialize runOutLabelsFlag = FALSE

   /*
    * Get a new label to use for a new object.
    */
   currentLabel++;

   /*
    * Does a label needs to be recycled?
    * Yes if the currentLabel is larger than MAX_NUM_LABELS or
    * runOutLabelsFlag is TRUE.  To recycle a label:
    * 1) Find the first alias and remove it from the list,
    * 2) relabel labeled object O so alias label is no longer used, and
    * 3) remove alias' entry from the alias list.
    * Note that this portion of the code should not be invoked until
    * SHORTINT_MAX labels have been used, implying that regions is not
    * an empty list.
    */
   if ((currentLabel > MAX_NUM_LABELS) OR
       (runOutLabelsFlag == TRUE)) {
      runOutLabelsFlag = TRUE        /* set flag first time */
      foundAlias       = FALSE       /* initialize alias found flag */
      currentObject    = *regions;
      ASSERT(currentObject != NULL) /* List should not be empty */

      while ((foundAlias != TRUE) AND      /* loop for alias not found */
             (currentObject != NULL)) {    /* and for each object */
         currentAlias = currentObject->aliasList;

         if (currentAlias != NULL){        /* First Alias found */
            foundAlias = TRUE              /* Set found flag */

            /*
             * Relabel labeled object in O replacing aliasLabel with the
             * objects' label.  Note that the bounding box may be larger
             * than the bounding box for just the alias.  It's possible
             * for the pixel requiring a new label to be within the
             * bounding box.  If this is true, don't relabel pixels to
             * the right of the pixel (they haven't been labeled yet).
             * For this particular case, it will only be the pixels on
             * the same row as pixel and to the right that have not been
             * labeled. The image is being traversed row-by-row and
             * column-by-column so pixels above and to the left of pixel
             * have been labeled and pixels below and to the right have
             * not been labeled.  Once the object image O has been
             * relabeled, the alias label will be returned to be reused.
             */
            currentLabel = currentAlias->aliasLabel
            aliasLabel   = currentAlias->aliasLabel
```

```
            objectLabel   = currentObject->labelValue
            get numColumns from image O
            pixelRow = floor(pixelLocation/numColumns)
            pixelColumn = pixelLocation - (numColumns * pixelRow)
            assert(currentObject->box.lowerRight.row <= pixelRow)
            if (pixelRow == currentObject->box.lowerRight.row) {
               rowMax = pixelRow - 1
               Loop for column = currentObject->box.upperLeft.column to
                                 pixelColumn
                  if (o(column,pixelRow) == aliasLabel) then
                     o(column,pixelRow) = objectLabel
               end loop for column
            } /* end if some unlabeled pixel in bounding box */
            else
               rowMax = currentObject->box.lowerRight.column

            Loop for row = currentObject->box.upperLeft.row to rowMax
               Loop for column = currentObject->box.upperLeft.column to
                                 currentObject->box.lowerRight.column
                  if (o(column,row) == aliasLabel) then
                     o(column,row) = objectLabel
               end loop for column
            end loop for row

            /*
             * Remove alias from list.
             */
            nextAlias                = currentAlias->nextAlias;
            currentObject->aliasList = nextAlias;
            currentAlias->nextAlias  = NULL;
            deleteAlias(currentAlias);
         } /* end if currentAlias */

         nextObject    = currentObject->nextObject;
         currentObject = nextObject;
      } /* end while currentObject */

      /* If there are no more aliases, return BACKGROUND_VALUE */
      if (foundAlias == FALSE)
         print warning "Max number of objects found, rest are ignored"
         currentLabel = BACKGROUND_VALUE

   } /* end need to recycle objects */

   return(currentLabel)
} /* end getLabel */
```

B.10.11 mergeObject

Name:	mergeObject	
Input:	*label*	label value to merge with given object
	background	label used for background pixels (non target)
	currentObject	object to merge with given label
	regions	linked list of objects

Output:	*currentObject* with label added as alias if not already alias *regions* linked list of objects, updated
Comment:	This routine determines whether *label* needs to be added to the alias list of a given object entry structure, *currentObject.* If *label* already exists in the definition of *currentObject*, nothing happens. If *label* doesn't exist, then the object containing *label* is added as an alias to *currentObject* (the two objects are merged together). The initial features of *currentObject* and the bounding box are updated to reflect this merge. If the object being merged (associated to *label*) has an alias list, this list is added to the alias list for *currentObject.* Assumptions: 1) *label* is not equal to the background value 2) *currentObject* != NULL
Calls:	createAlias() deleteObject() findObject()
Return:	MERGE_OBJECT_SUCCESS MERGE_OBJECT_ALLOC_ERROR

```
{ /* begin mergeObject */
#ifdef DEBUG
   if label == background then error
#endif DEBUG
   ASSERT(currentObject != NULL)

   /*
    * Is the label associated to currentObject (as it's label or
    * an alias of it's label)?  If yes, no merge required.
    */
   isLabel = FALSE

   if (currentObject->labelValue == label) /* is label THE label */
      isLabel = TRUE
   else {
      currentAlias = currentObject->aliasList
      while ((isLabel == FALSE) && (currentAlias != NULL)) {
         if (currentAlias->aliasLabel == label) /* label an alias? */
            isLabel = TRUE
         else {
            nextAlias    = currentAlias->nextAlias
            currentAlias = nextAlias
         } /* end else */
      } /* end while alias check */
   } /* end else */

   /*
    * The label is not already an alias/label of currentObject,
    * so merge objects:
    * 1) find the object associated to label, labelObject
    *    (then merge the objects currentObject and labelObject)
    * 2) add label to currentObject's alias list
    * 3) add labelObject's alias list to currentObject's alias list
    * 4) update the bounding box parameters to cover merged object, and
    * 5) combine the partial feature calculations to reflect merged
    *    object.
    */
```

```
if (isLabel == FALSE) {
   labelObject = findObject(*regions, label) /* find labelObject */
   ASSERT(labelObject != currentObject)

   /*
    * Update bounding box for object.
    */
   currentObject->box.upperLeft.column =
         MIN(currentObject->box.upperLeft.column,
             labelObject->box.upperLeft.column);
   currentObject->box.lowerRight.column =
         MAX(currentObject->box.lowerRight.column,
             labelObject->box.lowerRight.column);
   currentObject->box.upperLeft.row =
         MIN(currentObject->box.upperLeft.row,
          labelObject->box.upperLeft.row);
   currentObject->box.lowerRight.row =
         MAX(currentObject->box.lowerRight.row,
          labelObject->box.lowerRight.row);

   /*
    * Update initial feature values.
    */
   currentObject->initFeatures.centroidRow +=
      labelObject->initFeatures.centroidRow;
   currentObject->initFeatures.centroidColumn +=
      labelObject->initFeatures.centroidColumn;
   currentObject->initFeatures.area +=
      labelObject->initFeatures.area;
   currentObject->initFeatures.perimeter +=
      labelObject->initFeatures.perimeter;
   currentObject->initFeatures.mean +=
      labelObject->initFeatures.mean;
   currentObject->initFeatures.variance +=
      labelObject->initFeatures.variance;

   /*
    * Delete labelObject from the linked list of objects before
    * allocating newAlias (to minimize chance of an allocation
    * error).  Then store label for the labelObject as an alias
    * for currentObject.
    */
   aliasLabel = labelObject->labelValue
   aliasList  = labelObject->aliasList
   labelObject->aliasList = NULL
   call deleteObject(labelObject, regions)

   /* create alias list for labelObject (which is now an alias) */
   newAlias = createAlias()
   if (newAlias == NULL)
      call errorMessage("mergeObject: ")
      return MERGE_OBJECT_ALLOC_ERROR
   newAlias->aliasLabel = aliasLabel
   newAlias->nextAlias  = aliasList

   /*
    * Add labelObject's alias list to beginning of currentObject's
```

```
     * alias list.  If labelObject's alias list is not NULL, find
     * the end to add currentObject's alias list at the end.
     */
    nextAlias                  = currentObject->aliasList
    currentObject->aliasList = newAlias
    while (newAlias != NULL) {
       aliasList = newAlias
       newAlias  = aliasList->nextAlias
    } /* end if - find end of existing alias list */
    aliasList->nextAlias = nextAlias

  } /* end if merge objects */
  return MERGE_OBJECT_SUCCESS
} /* end mergeObject */
```

B.10.12 removeObject

Name:	removeObject	
Input:	*currentObject*	pointer to objectEntry structure to remove from linked list
	listHandle	pointer to linked list containing *currentObject*
Output:	*listHandle*	pointer to linked list without *currentObject* - different when *currentObject* was first element on list
Comment:	This routine removes an objectEntry, *currentObject*, from the linked list of objects. The linked list is updated so *currentObject* is no longer on the list and *currentObject* no longer points to any other objects.	
Calls:		
Return:	none	

```
{ /* begin removeObject */

   /*
    * Reset listHandle if object is the first element on the list.
    */
   if (currentObject == listHandle)
      listHandle = currentObject->nextObject;

   /*
    * Splice off object entry.
    */
   nextObject = currentObject->nextObject
   lastObject = currentObject->lastObject
   if (lastObject != NULL)
      lastObject->nextObject = nextObject
   if (nextObject != NULL)
      nextObject->lastObject = lastObject

   /*
    * Reset currentObject pointers from items on list to NULL.
    */
   currentObject->nextObject = NULL
   currentObject->lastObject = NULL

   return
} /* end removeObject */
```

B.10.13 selectSubset

Name:	selectSubset	
Input:	numObjects	number of objects on linked list *regions*
	selectNumber	number of objects to retain after ranking them
	regions	linked list of objects to rank and cull
Output:	*regions*	linked list containing selected subset of ROIs/objects
Comment:	This routine applies the ranking selection logic on the list of objects in *regions* to select a subset of the *selectNumber* objects in *regions* with the highest ranking metric (*mean*area*). If there are fewer objects in the list than desired, return the entire list without ranking them. If there are more objects than desired, build up a rankedList equal to the number desired, sorting the elements as you build the list. Then for additional candidates: compare with the smallest element on the rankedList. If the candidate has a smaller metric than the smallestRanked, delete the candidate. If the candidate has a metric larger than the smallestRanked, find the location to insert the candidate, delete the smallestRanked, reset the smallestRanked value. Repeat for the next candidate (until there are no more candidates). At any one time, only selectNumber of objects are kept on the rankedList. Assumptions: 1) numObjects >= 0 2) selectNumber > 0	
Calls:		
Return:	none	

```
{ /* begin selectSubset */

   assert(numObjects >= 0)
   assert(selectNumber > 0)

   /*
    * If less number of objects than desired, then don't bother
    * ranking objects, let all of them pass.
    */
   if (numObjects <= selectNumber)
      call errorMessage(
         "selectSubset: not many objects, no ranking done\n")
      return   /* objects NOT ranked */

   /*
    * Rank list of objects from highest to lowest ranking metric value
    *    nextObject points to higher value
    *    lastObject points to lower value
    */
   initialize rankedList to null
   initialize original list to regions
   /*
    * First initialize the list, build up list till you have
    * selectNumber objects: remove object from list, find place to
    * insert object, insert object
    */
```

```
for index = 1, selectNumber
   get currentObject from regions
   remove currentObject from regions
   find location on rankedList to insert currentObject
   insert currentObject onto rankedList
end for /* till list is full */

/*
 * List is full, now check each entry with smallest metric on list.
 * If the next entry has a smaller metric, discard.  Else find the
 * location to place the next entry on the list, insert the entry,
 * reset smallestObject, then continue.
 */
smallestObject = first object on rankedList
get currentObject from regions
while (currentObject != NULL) {      /* each object left on list */
   nextObject = currentObject->nextObject;
   removeObject(currentObject, regions)

   /*
    * case 1: current object metric <= smallest object metric
    *          delete object - have enough on list already
    */
   if (currentObject->rankMetric <= smallestObject->rankMetric) {
      deleteObject(currentObject, regions)
   }
   /*
    * case 2: smallest < current
    *          find place to insert in list, drop smallest
    */
   else {
      find place to insert in rankedList
      insert currentObject onto rankedList
      delete smallest element on rankedList
      reset smallest element of rankedList
   }
   /* go to next element */
   currentObject = nextObject;
 } /* end while there's an object on the list */

  return
} /* end selectSubset */
```

B.10.14 <u>updateObject</u>

Name:	updateObject	
Input:	backgroundValue	value/label associated to a background pixel
	amplitude image	image to use to calculate *mean* and *variance* features
	labeled image	image with labeled objects
	pixelLocation	single index associated to the pixel location to use to update initial features and bounding box
	thresholdLevel	above this value, pixels are "on target"
	currentObject	object that the pixel is being assigned to

Output:	*currentObject*	object with features and bounding box updated for pixel location given
Comment:	This routine incrementally updates the initial features and the bounding box for *currentObject,* the object that contains the pixel given. Assumptions: 1) amplitude image and labeled image have the same dimensions 2) pixelLocation is a single index corresponding to a valid coordinate in 2-dimensions, specifically the labeled image 3) the *currentObject* is the correct object associated to *pixelLocation* 4) the filtered image is being overwritten by the labeled image and the image is traversed top row to bottom row and left column to right column. 5) any pixel in the filtered image is considered a target pixel if the value is > thresholdLevel; otherwise, it is considered a background pixel. 6) there are no target pixels on the outer shell one pixel wide. Note that for the perimeter calculation, (determining whether the pixel is 8-adjacent to background pixel ?), since the target pixels cannot be on the outer shell of the image, the locations for the labeled image for an 8-adjacent neighbor are all valid offsets – no boundary cases occur – relative to location (column,row).	
Calls: (system)	floor	
Return:	none	

```
{ /* begin updateObject */

    assert(amplitude image and labeled image have same dimensions)
    assert(pixelLocation valid location in labeled image)
    /*
     * Determine row and column from pixelLocation.
     */
    row    = floor(pixelLocation / numColumns)
    column = pixelLocation - row * numColumns

    /*
     * Update bounding box for object.
     */
    currentObject->box.upperLeft.column  =
          MIN(currentObject->box.upperLeft.column, column);
    currentObject->box.upperLeft.row  =
          MIN(currentObject->box.upperLeft.row, row);
    currentObject->box.lowerRight.column  =
          MAX(currentObject->box.lowerRight.column, column);
    currentObject->box.lowerRight.row  =
          MAX(currentObject->box.lowerRight.row, row);

    /*
     * Update the initial features.
     */
    currentObject->centroidRow    += row;
    currentObject->centroidColumn += column;
    currentObject->area += 1;
    currentObject->mean     += amplitude(column,row);
```

```
currentObject->variance += (amplitude(column,row) *
                            amplitude(column,row));

/*
 * Is pixel 8-adjacent to background pixel ?
 * Since the target pixels cannot be on the outer shell of the image,
 * the locations for the labeled image below are all valid offsets
 * relative to location (column,row).  However, since the filtered
 * image is being overwritten by the labeled image, the pixels above
 * and to the left of the current pixel have been labeled, and the
 * pixels below and to the right of the current pixel have NOT been
 * labeled. To check for a perimeter pixel: if a labeled neighbor is
 * equal to the background value, or a non-labeled neighbor <=
 * thresholdLevel THEN the current pixel is a perimeter pixel. (There
 * are boundary conditions at the (numCols-2) column of the image
 * where the next pixel is already labeled as a background pixel; and
 * at the bottom row of the image where the pixels below have already
 * been labeled as background pixels.  Furthermore, any pixel on
 * these boundary conditions, are by definition, perimeter pixels
 * since they are next to a background pixel.)
 * Once it is determined that one of the 8-adjacent pixels is a
 * background pixel, the other adjacent pixels do not need to be
 * checked.
 */
perimeterFlag = FALSE
if ((labeled(column-1,row-1) == backgroundValue) ||
    (labeled(column,  row-1) == backgroundValue) ||
    (labeled(column+1,row-1) == backgroundValue) ||
    (labeled(column-1,row)   == backgroundValue))
   perimeterFlag = TRUE
check pixels to the right and below the pixel with the thresholdLevel
   setting the perimeterFlag to TRUE if any are > thresholdLevel
if (perimeterFlag == TRUE)
   currentObject->perimeter += 1;

return
} /* end updateObject */
```

B.11 FEATURE EXTRACTION MODULE

B.11.1 extractFeatures

Name:	extractFeatures	
Input:	*V* image	input image
	O image	labeled image
	distanceShort	short distance for GLCM descriptors
	distanceLong	long distance for GLCM descriptors
	regions	list of objects containing alias lists, bounding box, and
	initial	features for each object
Output:	*regions*	(NULL - objects are removed from this list and put on the
	list of	features)
	features	list of features containing entire set of feature values

Comment:	This routine is the highest level routine for the `Feature Extraction` module and provides an interface with the mainline. For each object on the *regions* list, a feature structure is created and the additional GLCM descriptors are calculated and stored on this **FeatureEntry** structure. This is a total of sixteen additional features (GLCM energy and entropy for each of two distances and four directions). The other features previously calculated are also available from this structure through a pointer. The *features* list has the same number of objects as the *regions* list and contains the entire set of features calculated for a selected object/ROI. Thus, the list of objects is emptied and a list of features is created where each entry on the feature list contains a single object from the region list. Assumptions: 1) *distanceShort* > 0 2) *distanceLong* > 0 3) *distanceShort* < number columns and number rows in *V* 4) *distanceLong* < number columns and number rows in *V* 5) *distanceShort* <= *distanceLong* 6) imageV and imageO are the same dimension
Calls:	calculateDescriptors() createFeature() removeObject()
(system)	
Return:	EXTRACT_FEATURES_SUCCESS EXTRACT_FEATURES_ALLOC_ERROR

```
{ /* begin extractFeatures */
   /*
    * Verify assumptions.
    */
   assert(distanceShort > 0)
   assert(distanceLong > 0)
   assert(distanceShort < number columns and number rows in V)
   assert(distanceLong < number columns and number rows in V)
   assert(distanceShort <= distanceLong)
   assert(imageV and imageO have the same dimensions)

   /*
    * Loop for each object found
    *    Loop for each distance, distanceShort and distanceLong
    *       Loop for each direction, 0, 45, 90, 135 degrees
    *          calculate GLCM energy and entropy
    *          add all feature values onto features
    *       end loop
    *    end loop
    * end loop
    *
    * Temporary memory is required to calculate the sum and difference
    * histograms.  Rather than have that memory allocated and freed for
    * each call to calculateDescriptors, have the arrays as static
    * memory in calculateDescriptors. (Or allocate it once here and
    * pass the buffers to calculateDescriptors to be used.)
    */
   initialize features to NULL
```

```
   object = *regions
   Loop for each object on regions list
      newitem = create featureEntry for output features
      if (error allocating memory for featureEntry) {
         call errorMessage("extractFeatures: error allocating memory\n")
         return EXTRACT_FEATURES_ALLOC_ERROR
      } /* end alloc error */
      Loop for distanceShort and distanceLong
         /* for 0 degree direction (horizontal) */
         dx = distance
         dy = 0
         call calculateDescriptors(V, O, object, dx, dy,
                                    &energy, &entropy)
         fill in descriptors for distance and 0° onto newitem
         /* for 45 degree direction (right diagonal) */
         dx = distance
         dy = distance
         call calculateDescriptors(V, O, object, dx, dy,
                                    &energy, &entropy)
         fill in descriptors for distance and 45° onto newitem
         /* for 90 degree direction (vertical) */
         dx = 0
         dy = distance
         call calculateDescriptors(V, O, object, dx, dy,
                                    &energy, &entropy)
         fill in descriptors for distance and 90° onto newitem
         /* for 135 degree direction (left diagonal) */
         dx = - distance
         dy = distance
         call calculateDescriptors(V, O, object, dx, dy,
                                    &energy, &entropy)
         fill in descriptors for distance and 135° onto newitem
      End loop for distances
      nextObject = object->nextObject
      removeObject(object,regions)
      newitem.objectPtr = object /* keep access to initial features */
      object     = nextObject
   End loop for objects

   return EXTRACT_FEATURES_SUCCESS
} /* end extractFeatures */
```

B.11.2 calculateDescriptors

Name:	calculateDescriptors	
Input:	*V* image	input image for histogram calculations
	O image	labeled image to define *object*
	object	**ObjectEntry** defining object
	distanceColumn	column distance (defining distance & direction)
	distanceRow	row distance (defining distance & direction)
Output:	energy	GLCM energy for given distance & direction
	entropy	GLCM entropy for given distance & direction

Comment:	This routine calculates the GLCM energy and entropy for a given object and a given distance. This is done be using the sum and difference histogram method described in [AAEC-1].
Calls:	isPixelObject()
(system)	
Return:	none

```
} /* begin calculateDescriptors */
   static array sumHistogram[NUM_HISTOGRAM_LEVELS],
                differenceHistogram[NUM_HISTOGRAM_LEVELS]

   initialize entropy and energy to 0.0

   if the distance cannot fit into the bounding box for the object
      return (no pixels at that distance in object, so values are 0)

   /*
    * Initialize the histogram arrays to zero.
    */
   initialize sumHistogram to 0
   initialize differenceHistogram to 0

   /*
    * Calculate the sum and difference histograms for an object, where
    * the histograms are incremented for each pixel that is part of the
    * object with a corresponding pixel (dx, dy) away that is also part
    * of the object.  For all pixels satisfying these conditions,
    * calculate the sum and difference and add to the histograms.
    * Calculate sumOffset and diffOffset which are offsets required to
    * guarantee that the bin index will be a positive number (within the
    * bounds expected).
    */
   totalNumPixels = 0
   Loop over bounding box for object
      If ((isPixelObject(x,y,0,object) AND
           (isPixelObject(x+dx,y+dy,0,object)) Then
         sumHistogram( [v(x,y) + v(x+dx,y+dy) + sumOffset] ) += 1
         differenceHistogram( diffOffset + v(x,y) - v(x+dx,y+dy) ) += 1
         totalNumPixels += 1
      Endif legal pixel address
   End loop for bounding box for object

   /*
    * Normalize sumHistogram and differenceHistogram and
    * calculate descriptors from sumHistogram and differenceHistogram
    */
   energyS = 0
   energyD = 0
   entropy = 0
   energy  = 0
   if (totalNumPixels > 0) {     /* there are values in histograms */
      Loop for i = 0, numHistogramLevels
         /*
          * If sumHistogram = 0 or differenceHistogram = 0, then there
          * is no contribution for that element in the histogram, only
          * positive values have a contribution.  Then never take
```

```
         * log(zero) so no need to check for this case.
         */
        if (sumHistogram(i) > 0) {
           sumHistogram(i) = sumHistogram(i) / totalNumPixels
           entropy = entropy - sumHistogram(i) * log(sumHistogram(i))
           energyS = energyS + sumHistogram(i) * sumHistogram(i)
        } /* end sumHistogram bin positive */
        if (differenceHistogram(i) > 0) {
           differenceHistogram(i) = differenceHistogram(i) /
                                        totalNumPixels
           entropy = entropy - differenceHistogram(i) *
                         log(differenceHistogram(i))
           energyD = energyD + differenceHistogram(i) *
                         differenceHistogram(i)
        } /* end differenceHistogram bin positive */
     End loop for i
     energy = energyS * energyD
   } /* end if there are values in histograms */

   return
} /* end calculateDescriptors */
```

B.11.3 createFeature

Name:	createFeature
Input:	none
Comment:	This routine allocates memory for a **featureEntry** structure which is used to store all the features calculated for each object or ROI found. The pointer members in this structure are initialized to NULL, the other members are not initialized.
Calls:	
(system)	malloc()
Return:	pointer to featureEntry

```
{ /* begin createFeature */
   allocate memory for featureEntry structure
   if alloc failed
      call errorMessage("createFeature: error allocating memory\n")
      return(NULL)

   /*
    * Initialize pointers to NULL.
    */
   newFeature->nextFeature = NULL
   newFeature->objectPtr   = NULL
   initialize entropy and energy values to a negative value (easily
      recognized on output as an error since these values are positive).

   return(newFeature)
} /* end createFeature */
```

B.11.4 deleteFirstFeature

Name:	deleteFirstFeature

Input:	pointer to feature list
Comment:	This routine frees memory associated to the featureEntry structure. It is assumed that a non-NULL valid pointer to a **FeatureEntry** structure is given. The memory associated to the first feature on the list is freed (After the **ObjectEntry** structure is freed) and the feature list pointer is updated.
Calls:	deleteObject()
(system)	free()
Return:	none

```
{ /* begin deleteFirstFeature */
   assert(featureList and *featureList)

   featureEntry = *featureList
   *featureList = featureEntry->nextFeature

   /*
    * Delete the ObjectEntry structure
    */
   if (featureEntry->objectPtr != NULL)
      deleteObject(featureEntry->objectPtr)
   free featureEntry structure

   return
} /* end deleteFirstFeature */
```

B.11.5 isPixelObject

Name:	isPixelObject	
Input:	column	column coordinate of pixel to test
	row	row coordinate of pixel to test
	O	labeled image *O*
	object	**ObjectEntry** structure for object in question
Comment:	This routine checks a pixel location in labeled image *O* to determine whether that location is part of the object represented by objectEntry *object*. The return value is IS_PIXEL_OBJECT_TRUE if the pixel is part of the object (i.e., has a label equal to the object's label or an alias of the object's label). Otherwise, an IS_PIXEL_OBJECT_FALSE is returned. Assumptions: 1) *O* is not NULL 2) column and row are within the bounds of the *O* image 3) object is not NULL	
Calls:		
(system)		
Return:	IS_PIXEL_OBJECT_TRUE IS_PIXEL_OBJECT_FALSE	

```
{ /* begin isPixelObject */
   assert(O != NULL)
   assert(column and row are within bounds of O dimensions)
   assert(bounding box for O is valid
         (fits within the dimensions of imageO))
```

```
   /*
    * Initialize return value to FALSE and get the label value
    * at the coordinate of interest.
    */
   foundFlag = IS_PIXEL_OBJECT_FALSE

   /*
    * If the pixel location is outside of the object's bounding box,
    * then the pixel is definitely NOT part of the object.
    */
   if column and row outside bounding box
      return(IS_PIXEL_OBJECT_FALSE)

   /*
    * Check object's label first, then check aliases.
    */
   labelValue = o(column, row)
   if (object->labelValue == labelValue) { /* found label */
      foundFlag = IS_PIXEL_OBJECT_TRUE
      }
   else { /* check aliases */
      currentAlias = object->aliasList
      while ((! foundFlag) && (currentAlias != NULL)) {
         if (currentAlias->labelValue == labelValue) { /* found label */
            foundFlag = IS_PIXEL_OBJECT_TRUE
         }
         nextAlias    = currentAlias->nextAlias
         currentAlias = nextAlias
      } /* end while loop for aliases */
   } /* end else check aliases */

   return(foundFlag)
} /* end isPixelObject */
```

B.12 RETURN CODES

This section lists all of the return codes for each function listed in the previous sections. Note that the codes are prefixed by the name (possibly abbreviated) of the function. The return code from the mainline is DIS_IMAGE_UNDERSTANDING_SUCCESS (FALSE), and is DIS_IMAGE_UNDERSTANDING_ERROR (TRUE).

```
ADD_OBJECT_SUCCESS
ADD_OBJECT_ALLOC_ERROR

DILATE_AND_SUBTRACT_SUCCESS
DILATE_AND_SUBTRACT_ALLOC_ERROR

EXTRACT_FEATURES_SUCCESS
EXTRACT_FEATURES_ALLOC_ERROR

FILTER_IMAGE_SUCCESS
FILTER_IMAGE_ALLOC_ERROR
```

```
FIND_CONNECTION_SUCCESS
FIND_CONNECTION_ALLOC_ERROR

IS_PIXEL_OBJECT_TRUE
IS_PIXEL_OBJECT_FALSE

MERGE_OBJECT_SUCCESS
MERGE_OBJECT_ALLOC_ERROR

READ_INPUT_SUCCESS
READ_INPUT_ALLOC_ERROR
READ_INPUT_INVALID_INPUT_PARAMETER
READ_INPUT_INVALID_KERNEL
READ_INPUT_KERNEL_NOT_ODD
READ_INPUT_KERNEL_TOO_LARGE
READ_INPUT_OPEN_FILE_ERROR
READ_INPUT_READ_FILE_ERROR
READ_INPUT_CLOSE_FILE_ERROR

READ_SHORT_INT_IMAGE_SUCCESS
READ_SHORT_INT_IMAGE_ALLOC_ERROR
READ_SHORT_INT_IMAGE_INVALID_DIMENSIONS
READ_SHORT_INT_IMAGE_READ_ERROR
READ_SHORT_INT_IMAGE_INVALID_DATA

READ_UCHAR_IMAGE_SUCCESS
READ_UCHAR_IMAGE_ALLOC_ERROR
READ_UCHAR_IMAGE_INVALID_DIMENSIONS
READ_UCHAR_IMAGE_READ_ERROR

SELECT_REGIONS_SUCCESS
SELECT_REGIONS_ALLOC_ERROR

WRITE_OUTPUT_SUCCESS
WRITE_OUTPUT_OPEN_FILE_ERROR
WRITE_OUTPUT_WRITE_FILE_ERROR
WRITE_OUTPUT_CLOSE_FILE_ERROR
If
/*
 * Done
 */
Return
```

Appendix C: DIS Benchmark C Style Guide

C.1 PURPOSE

The purpose of this document is to define one style and set of standards and guidelines for generating C language software for the Data-Intensive Systems (DIS) Benchmarking Project. It is intended that software developed according to this style and set of standards be correct and easy to maintain. In order to attain these goals, the software should:

have a consistent style;

be easy to read and understand;

be portable to other architectures;

be free of common types of errors; and

be maintainable by different programmers.

Throughout this manual the rules of style will be referred to as either standards or guidelines. A *standard* is an absolute rule which should always be followed. A *guideline* is a rule which should always or almost always be followed, but it either requires that a qualitative judgment be made or there are a few cases where it would be acceptable (or even encouraged) to break the guideline.

For example, the fact that a copyright notice must be included in every program is a standard. This rule should never be broken. The rule that functions really may not be longer than two pages in length is a guideline –there is no requirement that every function contain exactly some number of lines, but the programmer needs to divide up a function into smaller parts when it starts to get "too long".

C.2 SCOPE

The scope of this document is C coding style; questions of design and functional organization are beyond the scope of this document. Books on these subjects are readily available elsewhere. Of necessity, these standards cannot cover all situations.

Coding styles for other languages used for DIS benchmarking, if any, will be developed separately, but will be as consistent with this C style guide as possible for ease of transition.

C.3 FILE ORGANIZATION

C.4 STANDARD: FILE-NAMING CONVENTIONS	The following file-naming conventions will be used: • C source code files — *.c • header files — *.h • make — Makefile • shared library — *.so

C.5 STANDARD: ALL SOURCE CODE FILES MUST CONTAIN THE APPROPRIATE COPYRIGHT NOTICE	Each source file must have one of the following copyright statements at the beginning of the file.
	`/* Copyright 1999, Atlantic Aerospace Electronics Corp. */`
	`/* Copyright 1999, The Boeing Corporation */`
	`/* Copyright 1999, ERIM International, Inc. */`

C.6 GUIDELINE: FUNCTION LENGTH < 2 PAGES	If functions are longer than about 2 pages, then the design of the function should be questioned. Especially consider if more than one function is involved or if sub-functions would be better put into separate modules. When considering the length of the function, comment blocks should not be included.
C.7 GUIDELINE: ONLY ONE FUNCTION PER SOURCE FILE	Under normal circumstances, there should only be one function per source code file. In certain cases, very short and tightly related functions may be grouped together in one file.
C.8 STANDARD: LINE LENGTH < 80 CHARACTERS	Lines of code must be less than 80 characters in length. Long lines must be broken into smaller pieces, such that when the file is printed, all portions of the code will print out legibly. It is recommended that subsequent sections of a longer line be indented so that it is clear that these are continuations of the line above. The length of a line includes any commentary that follows the actual C code on the line, and, as well, any white space that precedes the code as indentation.
C.9 STANDARD: SECTIONS MUST BE ORDERED THIS WAY	The order of sections for a program, or source code, file is as follows: 1. **Prologue:** First in the file is a prologue that tells what is in that file. A description of the purpose of the objects in the files (whether they be functions, external data declarations or definitions, or something else) is more useful than a list of the object names. A description of the method(s) used is helpful for any complex function. Descriptions should not be so detailed that maintenance of the prologue takes more effort than is gained by increased understanding of the code itself. 2. **Includes:** Any header file includes should follow the prologue. If the include is for a non-obvious reason, the

reason should be commented. In most cases, system include files like stdio.h should be included before user include files.

3. **Defines:** Any defines that apply to the file as a whole are after the includes.

4. **Global Definitions:** After the defines come the global (external) data declarations.

5. **Functions:** The functions come last and, if there is more than one, should be in some sort of meaningful order.

C.10 GUIDELINE: HEADER FILE ORGANIZATION

All functions in a given header file should be related to the same general function, i.e. declarations for separate sub-systems should be in separate header files.

Example: all functions in the header file stdio.h either perform or assist in the performance of input and output.

C.11 GUIDELINE: HEADER FILE ORGANIZATION

Within a header file, functions that perform related tasks should be grouped in the same section.

Example: within the file stdio.h, all functions in the scanf family should be placed together.

C.12 GUIDELINE: HEADER FILE-NAMING

Avoid private header filenames that are the same as public header filenames.

C.13 GUIDELINE: HEADER FILE SPECIFICATION

Don't use absolute pathnames for header files. Use the <name> construction for getting them from a standard place, or define them relative to the current directory.

C.14 GUIDELINE: HEADER FILE SPECIFICATION

The "include-path" option of the C compiler (-I on many systems) used in the Makefile is the best way to handle extensive private libraries of header files; it permits reorganizing the directory structure without having to alter source files.

C.15 STANDARD: HEADER FILE INCLUSION IN THE FILE THAT DEFINES THE FUNCTION

Header files that declare functions or external variables must be included in the file that defines the function or variable. That way, the compiler can do type checking and the external declaration will always agree with the definition.

C.16 STANDARD: HEADER FILE INCLUSION IN THE FILE THAT DEFINES THE FUNCTION

Put code like the following into each .h file to prevent accidental double-inclusion.

```
#ifndef EXAMPLE
#define EXAMPLE
...    /* body of example.h file */
#endif /* EXAMPLE */
```

Generally, the macro name should be the file name, all in capital letters, with words separated by underscores, periods replaced by underscores, and prefixed by DIS_.

Example: dataBase.h would use DIS_DATA_BASE_H.

C.17 GUIDELINE: HEADER FILE NESTING

Header files should contain all the necessary elements needed for its own compilation. Therefore, header files may be nested.

C.18 VARIABLE DECLARATIONS

C.19 STANDARD: VARIABLE NAMING

All variable names must begin with a lowercase letter.

The words of a compound variable name should be separated by capitalizing the first letter of every word except the first word, e.g. someVariable. In some cases where it is clearer, variables may be named by separating words with underscores, e.g. some_variable.

Function names are variable names, and should be assigned accordingly.

No special provision is made for pointers, globals, et cetera. The name and usage should be so clear that special prefixes or suffixes would only confuse, rather than clarify.

C.20 STANDARD: TYPE NAMING

Type names must begin with an uppercase letter.

The words of a compound type name should be separated by capitalizing the first letter of every word, e.g. SomeType. In rare cases where it is clearer, types may be named by separating words with underscores, e.g. some_type.

C.21 STANDARD: MACRO NAMING

Macro names must be all upper case separated by underscores, e.g.

```
#define WEEK_DAYS 7
```

C.22 GUIDELINE : DO NOT USE GLOBAL VARIABLES

The use of global variables is strongly discouraged. It is conceivable, however, that there will be cases where the use of global variables can actually make a program more readable by not cluttering function calls. Any use of global variables by a function should be documented in the prologue.

C.23 STANDARD: GLOBAL VARIABLES DECLARATION AT TOP OF FILE

Any global variables must be declared at the top of a file, before any function declarations. Variables which are global to only the functions in a single file should be declared as "static".

C.24 GUIDELINE :LOCAL VARIABLE DECLARATION

Variables should be declared with as local a scope as possible.

Note: C permits the declaration of variables within any block, e.g., within a "for" block; however declaration must be at the beginning of the block.

C.25 STANDARD: USE SYMBOLIC CONSTANTS INSTEAD OF LITERALS

All quantities that must remain unchanged throughout a program must be named using either the "#define" macro capability or a “const” type variable. For example:

```
#define MAX_CHANNELS 4096
const float pi 3.14159
```

The former is preferred in cases where use of the latter would result in a global variable.

C.26 STANDARD: MACRO DEFINITION	Macros that are used only in a single compiland should be defined at the beginning of that compiland. Macros that have a more general scope should be defined and documented in header files of appropriate scope.
C.27 GUIDELINE : NON-STATIC VARIABLE INITIALIZATION NOT IN DECLARATION	Non-static variables must not be initialized in their declarations. Instead, give a variable its initial value in a separate statement line just before the variable is first used. This is because initialization within the declaration statement only occurs once under the language definition of Kernigan & Ritchie; if the function is used more than once the variable will not be re-initialized. The ANSI language definition requires that automatic variables initialized in their declarations be reset every time the function is entered, but we may not be able to guarantee the use of an ANSI compiler.
C.28 STANDARD: EXPLICIT +1 IN STRING LENGTH DECLARATION FOR \0	Character arrays used as strings, i.e., to hold ASCII text and terminated by a null character) should have a defined length that explicitly includes the "+ 1" character for the null string terminator.

```
#define NAME_LEN 20 + 1
char name[NAME_LEN];
```

C.29 STANDARD: STRUCTURE DECLARATION	Each field in a structure must be declared on a separate line. The structure must have a tag, named with the same format as a type. The actual assignment of a structure to a variable must be done in a separate statement.

```
typedef struct {
    char name[NAME_LEN];
    char author[AUTHOR_LEN];
    long number;
} Book;

Book goneWithTheWind;
```

C.30 STANDARD: USE CONVENTIONAL ABBREVIATIONS FOR COMMON VARIABLES

• average	avg
• database	db
• data-intensive system	dis
• frequency	freq
• image	img
• length	len
• message	msg
• number	num
• pointer	ptr
• position	pos
• string	str

C.31 GUIDELINE : MACROS	Include a comment on the same line as macro declarations.
C.32 GUIDELINE	Place all macro definitions at the beginning of the file after the

: MACROS	prologue, or in a header file.
C.33 GUIDELINE : MACROS	Place all shared macros in a header file.
C.34 GUIDELINE : MACROS	Place parentheses around the parameters in the replacement text and around the entire text whenever possible. For example: `#define NEXT(p) ((p)->_next)`
C.35 FUNCTIONS	
C.36 STANDARD: PARAMETER LIST	Each parameter passed to a function should appear on a separate line in the function declaration, with a short comment describing its function.
C.37 STANDARD: PARAMETER LIST INDENTATION	If the function and its parameter list is longer than one line, lines after the first one will be indented from the left margin so that the second line of the parameter list starts directly below where the parameter list begins on the first line.
C.38 STANDARDS: RETURN VALUES MUST BE EXPLICITLY DECLARED	Return values must be explicitly declared. The function declaration should return void if an actual value is not being returned.
C.39 STANDARD: RETURN VALUES	A function that returns information via one or more of its parameters may return only status information in its name.
C.40 STANDARD: PARAMETER ORDER	The list of function parameters should have a definite order. The standard for the C library functions is opposite from most other languages and from most programmer's intuition. For example, the string copy function, strcpy(out, in), takes the output parameter first and then the input parameter. This makes sense if you think of the ordering like you think of the ordering of an assignment statement. We can't change the C library standard but for routines which we write, we will use the more conventional parameter ordering of input, modified (input and output), and finally output parameters.
C.41 STANDARD: STATIC FUNCTION	Any function which is only called from other functions in the same file should be declared "static".
C.42 TEMPLATE FOR FUNCTION DECLARATIONS	(see code below)
C.43 STANDARDS:	The rest of the function, until the closing brace will be indented one

```
void funcName (int firstParameter, /*Comment 1st
param*/
               int secondParameter,   /*Comment 2nd
param*/
               double thirdParameter, /* ... */
               char fourthParameter)  /* ... */
{ /* beginning of funcName() */
    return;
} /* end funcName() */
```

FUNCTION BODY	step (two or three spaces - do not use tabs). A beginning brace, "{", opening the body of the function must be on a line by itself (with comment) and left justified or at the end of the line which introduces the block. Of course, any control statements will cause further indentation from this basic indentation.
C.44 STANDARDS: FUNCTION BODY	A single return statement must always be present with a parameter if the function is not of type void. Even if the function has no return value and, therefore the C language does not require a return, it is good practice to use one. This can be useful for setting a break-point during debugging.
C.45 STANDARDS: FUNCTION BODY	A closing brace, "}", closing the body of the function must be on a line by itself (with comment) and left justified.
C.46 STANDARDS: FUNCTION PROTOTYPE	Function prototypes must be generated for all functions generated. The prototypes must include all variable types and names.

C.47 COMMENTS

At one extreme, it is obvious that a program of any significant complexity that has no comments is hard to read and maintain. At the other extreme, a program that is loaded with comments can get tedious to read. If the comments are scattered around in the code with excessive white space, the amount of code that can be seen on a typical 25-line monitor is insufficient for effective viewing and/or editing. Therefore, it is recommended that:

C.48 GUIDELINE: TEMPLATE FOR FILE AND FUNCTION HEADER

If a file contains functions which are closely enough linked that when one is modified it will probably be necessary to modify all or most of the functions in the file, then the following header should be placed at the beginning of the file and a normal block comment briefly describing the function placed at the beginning of each individual function.

```
/* Copyright 1999, Atlantic Aerospace Electronics Corp
*/
/*
 * File Name:
 * Purpose:
 * Documentation: (if appropriate)
 *
 * Revision History:
 * Date      Name          Revision
 * ------- ------------  ----------------------------
 * 02Jan99 Simon Birch   Created
 */
```

C.49 GUIDELINE: FUNCTION COMMENTS

Comments inside a function body should occur in only two forms: block comments and line comments.

C.50 GUIDELINE: BLOCK COMMENTS

Block comments should precede cohesive blocks of code, and describe the block fully. Code is more readable when comments are presented in paragraph form prior to a block of code, rather than a line of comment for a line or two of code. The idea is to not obscure the readability of the code by interspersing comment lines. Comments are most appropriate for cohesive blocks of code when they explain the purpose of the block in accomplishing a cohesive task. This enables the reader to understand the function being implemented. Thus the reader will be able to quickly find the appropriate section of code without getting bogged down in coding details for other sections of code.

C.51 STANDARDS: BLOCK COMMENTS

Comment blocks should be preceded by a single blank line. Multiple-line, block comments must have an asterisk character, "*", at the beginning of each line of the block (after the first one which, of course, must start with "/*"). The last line of the block comment is "*/".

```

/*
 * This is an example of a block comment. Note the
opening
 * and closing lines, and the vertical alignment of the
 * asterisks on each text line. Also note that there
will
 * be one blank line before the comment block.
 */
```

C.52 STANDARDS: BLOCK COMMENTS

Indent block comments to the same level as the block being described.

C.53 GUIDELINE: LINE COMMENTS	Line comments can augment, but not replace, block comments. They should serve as special clarification on lines within a block to make following the code simpler. They are especially useful for commenting on variable and macro definitions and the like. They should be left-justified at a column to the right of the code for easier readability. In exception to this rule, comments for brackets should be kept with the brackets.

```
...
    if (x < y) {
        x = doSomething();            /* x=0 on failure */
        y = doSomethingElse(z);       /* assumes z<65536
*/
    } /* end of if (x < y) */
```

C.54 STANDARD: BRACKETS	No open- or close-bracket ('{' or '}') shall appear on a line by itself without a comment identifying the block being opened or closed. For example:

```
for (l = 0; l < 10; l++) {
    x = x + 1;
} /* end of loop for l */
```

C.55 GUIDELINE: FUNCTION COMMENTS	Meaningful variable and function names can minimize the need for comments. Using this approach decreases the chance of having the code change but an associated comment not be updated to reflect the code change.

C.56 STATEMENTS

C.57 STANDARD: ONE STATEMENT PER LINE	Every statement must begin on a separate line. "a = 2; b = 3;" is not allowed.
C.58 GUIDELINE: DO NOT USE GOTO AND CONTINUE STATEMENTS	"Goto" and "continue" statements are highly discouraged. The use of "break" commands is discouraged except in "switch" statements (where they are required).

```
status = SUCCESS;
i = 0;
while ((string[i] != NULL) && (status ==SUCCESS)) {
    transmitCharacter(string[i], &status);
} /* end of while */
```

is preferred over:

```
i = 0;
while (string[i] != NULL) {
    transmit_character(string[i], &status);
    if (status != SUCCESS) break;
} /* end of while */
```

C.59 GUIDELINE: EXIT STATEMENT DISCOURAGED	Explicit use of the "exit();" statement is discouraged, except for error-handling functions.

C.60 GUIDELINE: MAXIMUM 4 LEVELS OF CONTROL STRUCTURE NESTING

Nesting of statements, "if", "for", "while", etc., should go no more than 4 levels. If more levels appear to be needed, consider use of a function at one of the higher levels.

C.61 STANDARDS: NULL STATEMENTS

Null statements must include a comment line. For example, if the "default:" case in a switch statement does nothing, put in a "default:" label followed by a comment to the effect: /* no action */ followed by the break statement.

C.62 STANDARD: COMMENT FOR MISSING "BREAK" IN SWITCH

If a particular case in a switch statement is meant to drop through to the next case (i.e., it has the same effect), the fact that the earlier case has no "break" statement should be explicitly noted with a comment.

```
switch(whichIsotope) {
    case PU239 : /* same action as for PU240; no break
*/
    case PU240 :
        processPlutoniumIsotope();
        break;
    default :
        break; /* if not PU 239 or 240, no action taken
*/
} /* end of switch(whichIsotope) */
```

C.63 STANDARD: STATEMENT BLOCKS

Statements that affect a block of code (i.e., more than one statement) must either have the opening brace "{" at the end of the line containing the control statement, or the opening brace must be on the line immediately below and lined up with the first letter of the control statement.

C.64 STANDARD: STATEMENT BLOCKS

The body of the block must be indented one step from the control statement.

C.65 STANDARD: STATEMENT BLOCKS

The ending brace, "}", must be on a line by itself (with comment) and at the same indentation level as the control statement.

C.66 EXAMPLE: STATEMENT BLOCKS

```
Examples: if (first < last) {
    resultOne = first / 2;
    resultTwo = last / 2;
} /* end of if */
else {
    resultOne = last / 2;
    resultTwo = first / 2;
} /* end of else */
do
{   status = doSomething();
} while (status == SUCCESS);
```

C.67 GUIDELINE: "BLOCK" WITH SINGLE STATEMENT

The programmer is encouraged to block off even a single statement following a "while", "if", "else", etc. Using the curly braces "{}" is required only when there is a block of more than one statement. However, putting in the braces makes the scope of the control

STATEMENT	statement very clear and helps to protect the code in the event that a second line is added to the block if the single line contains a macro which translates into more than one line of code.

```
if (someCondition == TRUE) {
    thisVariable = thatVariable;
}
```

C.68 STANDARD: INCREMENT AND DECREMENT OPERATOR ONLY AS POSTFIX	The increment and decrement operators, "++" and "--", are permitted only in post-fix notation, e.g., "i++".
C.69 GUIDELINE: INCREMENT AND DECREMENT OPERATORS NOT IN OTHER STATEMENTS	The increment and decrement operators, "++" and "--", should not appear as a part of any other statement.

```
while (string[i] != NULL) {
    doSomething();
    I++;
}
```

is preferred over

```
while (string[i++] != NULL) {
    doSomething();
}
```

C.70 GUIDELINE: BRANCHING ON FUNCTION CALL	Often a program will branch based on the success or failure of a function call. It is clearer to break out the function call onto a separate line followed by a new line containing the conditional statement:

```
ptr_FileHandle = fopen("some_file", READ_ONLY);
if (ptr_FileHandle == NULL) {
    printf("Could not open file; program
terminating.");
    terminateApplication();
}
else {
    doSomething();
}
```

is easier to understand than:

```
if ((fileHandle = fopen("some_file", READ_ONLY))==
NULL) {
    printf("Could not open file; program
terminating.");
    terminateApplication();
}
else {
    doSomething();
}
```

Generally speaking, if the return value is to be stored and utilized, the call and the conditional should be separate statements. However, note that when the return value is not needed after the conditional, the function call may be issued within the conditional without a “compound statement”.

The "fopen" example above demonstrates a style which helps to avoid internal side effects. "Side effect" as used in this example refers to the fact that the reader may focus on the "if(xxx == NULL)" aspect of the statement and not fully realize that there is a call to "fopen()". The other danger of this type of compound statement is the potential for completely changing the meaning if the parentheses around the "fileHandle = fopen()" part are left off.

C.71 EXAMPLE TO EMPHASIZE POTENTIAL PROBLEMS OF "COMPOUND" CONDITIONAL STATEMENTS

```
if (((fileOne = fopen("some_file", READ_ONLY)_ != NULL)
&&
    ((fileTwo = fopen("other_file", READ_ONLY)) !=
NULL)) {
    doWhatEver();
}
```

would be much easier to understand if written as follows:

```
fileOne = fopen("some_file", READ_ONLY);
if (fileOne != NULL) {
    fileTwo = fopen("other_file", READ_ONLY);
    if (fileTwo != NULL) {
        doWhatEver();
    }
}
```

If the evaluation of the first half of the compound "and" fails, the second part is not evaluated. Thus, this type of statement is not very clear as to what really happens. The second example makes the action much clearer.

C.72 GUIDELINE: AVOID INTERNAL SIDE EFFECTS

Expressions should not produce any internal side effects. Statements like "while (string[i++] != 0)" should be avoided.

C.73 GUIDELINE: COMPOUND FOR-LOOP OPERATORS

C allows compound statements within the initialization and iteration expressions of for-loops. All statements within these expressions must be directly relevant to the operation of the loop, and not simply to operations that need to take place within or during initialization of the loop block.

C.74 STANDARD: DO NOT USE DEFAULT TRUTH VALUE TEST

Do not default the test for non-zero, i.e.

```
if (f() != FAIL)
```

is better than

```
if (f())
```

even though FAIL may have the value 0 which C considers to be false. An explicit test will help you out later when somebody decides that a failure return should be -1 instead of 0. Use explicit comparison even if the comparison value will never change.

C.75 OPERATORS

C.76 STANDARD: USE SINGLE SPACES AROUND BINARY OPERATOR

All operators which take two parameters must have a single space on either side of the operator. This makes it very handy to use an editor to search for a variable assignment; you need only search for "a =" and not "a =" as well as "a=". It also makes the code more readable.

C.77 STANDARD: NO SPACE

In contrast to binary operators, all unary operators (e.g., a minus sign or the address operator, "&") should not have a space between the

FOLLOWING UNARY OPERATOR

operator and the object.

C.78 GUIDELINE: CONDITIONAL OPERATOR DISCOURAGED

The use of the ternary conditional operator, "?:" in the main program is discouraged, primarily to make the code more readable. It can still be used in macros.

```
if (abc > xyz) {
    zOne = abc;
}
else {
    zOne = xyz;
}
```

is generally easier to follow than:

```
zOne = (abc > xyz) ? abc : xyz;
```

C.79 STANDARD: USE PARENTHESES TO REMOVE PRECEDENCE AMBIGUITY

Where operator precedence must be known to determine the meaning of an expression, it is required that you use parentheses to eliminate any ambiguity which might arise from lack of knowledge of operator precedence. For example, to increment the variable pointed to by the pointer "imageData", use "(*imageData)++". This use of parentheses makes it clear that the contents of location "imageData" is being incremented and not the address itself.

C.80 WHITE SPACE

C.81 GUIDELINE : USE OF WHITESPACE FOR READABILITY

Use vertical and horizontal whitespace judiciously to make the program more readable. Indentation and spacing should reflect the block structure of the code.

C.82 STANDARD: VERTICAL SPACING OF CONDITIONAL OPERATORS ON SEPARATE LINES

A long string of conditional operators should be split onto separate lines.

```
if (foo->next==NULL && totalCount<needed
    && needed<=MAXLLOT && serverActive(currentInput)) {
...
```

will be better as

```
if (foo->next == NULL
    && totalCount < needed
    && needed <= MAXLLOT
    && serverActive(current -input))
{...
```

Similarly, elaborate "for" loops should be split onto different lines.

```
for (curr = *listp, trail = pList;
     curr != NULL;
     trail = &(curr->next), curr = curr->next) {
    doSomething();
    doAnotherSomething();
}
```

C.83 STANDARD: SPACING FOR PARENTHESES

Keywords that are followed by expressions in parentheses should not be separated from the left parenthesis. Also put blanks after commas in argument lists to help separate the arguments visually.

C.84 STANDARD: NO TAB CHARACTERS	Source code files should use only space characters for horizontal whitespace. TAB markers should be converted to spaces before delivery.

C.85 CONSTANTS

C.86 STANDARD: NAMING SYMBOLIC CONSTANTS IN UPPER CASE	Symbolic constants must be in upper case. e.g., "TRUE".
C.87 GUIDELINE : CONSISTENCY OF CONSTANT DEFINITIONS	Constants should be defined consistently with their use; e.g. use 540.0 for a double instead of 540 with an implicit float cast.

C.88 CONDITIONAL COMPILATION

C.89 GUIDELINE : USE OF CONDITIONAL COMPILATION	Conditional Compilation should only be used for controlling the compilation of machine-dependent code, setting optimizations at compile time, and debugging.
C.90 STANDARD: DEFAULT OF CONDITIONAL COMPILATION	A conditional compiland controlling machine-dependent code should default to an error. The default for a conditional compiland meant to allow for code optimization should be un-optimized code.

C.91 PORTABILITY

C.92 GUIDELINE : MACHINE-DEPENDENT CODE USE	Only write machine-dependent code when necessary. Even if, for example, a particular piece of hardware requires that a machine-dependent routine be written, try to write any routines that support the machine-dependent code machine-independently.
C.93 STANDARD: ANSI-COMPATIBLE CODE	All code should conform to the standard established by ANSI.
C.94 STANDARD: MACHINE-DEPENDENT CODE IN SEPARATE FILE	Place all machine-dependent code in a separate file from all machine-independent code.
C.95 STANDARD: MACHINE-DEPENDENT CODE IN SEPARATE FILE	Machine-dependent code must be #ifdef'ed so that an informative error message will result if the code is compiled on a machine other that which it is designed for.

C.96 GUIDELINE : PORTABILITY	Try not to assume ANSI. Even though ANSI-compliant code should be written, if you know that many non-ANSI compilers will not understand some code and you know of a more portable way of writing the code which is still ANSI-compatible, do so.
C.97 GUIDELINE : PORTABILITY	Be aware that the size of different data types may vary from platform to platform. Be especially careful to avoid making assumptions about integers and pointers.
C.98 GUIDELINE : PORTABILITY	Also be aware that the precision and storage format of floating point numbers may vary from platform to platform.
C.99 GUIDELINE : PORTABILITY	Do not assume that software will always be executed on the machine for which it is originally designed.

C.100 MISCELLANEOUS

C.101 STANDARD: ALWAYS CHECK FUNCTION RETURN VALUE	Always check the return values of functions which return a special value when an error occurs. If there really were no chance of an error occurring, then the function would not have such a return value.
C.102 GUIDELINE : USE LIBRARY FUNCTIONS WHEN POSSIBLE	Use functions in libraries whenever possible instead of "re-inventing the wheel".

C.103 CONCLUSION

The standards and guidelines laid forth in this document should be followed zealously and in good faith (do not, for example, ignore guidelines just because they do not say must). Remember above all that you are working on a team of programmers, and should therefore labor to make your code as easy for another to follow as possible in case another person has to modify your program. It is not unheard of for even the person who writes sloppy code to not be able to follow it after a significant amount of time has passed.

C.104 REFERENCES

1. Los Alamos National Laboratory, *C Style and Coding Standards for the SDM Project*, August 15, 1996.
2. Spencer, Keppel, and Brader, *Recommended C Style and Coding Standards*, Revision 6.0, June 25, 1990.
3. The Boulder Software Group, *Example C Style Guidelines*, 1990.

☆U.S. GOVERNMENT PRINTING OFFICE: 2000-610-055-10030

DISTRIBUTION LIST

addresses	number of copies
CHRISTOPHER J. FLYNN AFRL/IFTC 26 ELECTRONICS PKWY ROME NY 13441-4514	1
SM&A CORPORATION 1300-B FLOYD AVE ROME NY 13440-4600	2
AFRL/IFOIL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	1
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
ATTN: NAN PFRIMMER IIT RESEARCH INSTITUTE 201 MILL ST. ROME, NY 13440	1
AFIT ACADEMIC LIBRARY AFIT/LDR, 2950 P.STREET AREA B, BLDG 642 WRIGHT-PATTERSON AFB OH 45433-7765	1
AFRL/HESC-TDC 2698 G STREET, BLDG 190 WRIGHT-PATTERSON AFB OH 45433-7604	1

ATTN: SMDC IM PL 1
US ARMY SPACE & MISSILE DEF CMD
P.O. BOX 1500
HUNTSVILLE AL 35807-3801

TECHNICAL LIBRARY D0274(PL-TS) 1
SPAWARSYSCEN
53560 HULL ST.
SAN DIEGO CA 92152-5001

COMMANDER, CODE 4TL000D 1
TECHNICAL LIBRARY, NAWC-WD
1 ADMINISTRATION CIRCLE
CHINA LAKE CA 93555-6100

CDR, US ARMY AVIATION & MISSILE CMD 2
REDSTONE SCIENTIFIC INFORMATION CTR
ATTN: AMSAM-RD-OB-R, (DOCUMENTS)
REDSTONE ARSENAL AL 35898-5000

REPORT LIBRARY 1
MS P364
LOS ALAMOS NATIONAL LABORATORY
LOS ALAMOS NM 87545

ATTN: D'BORAH HART 1
AVIATION BRANCH SVC 122.10
FOB10A, RM 931
800 INDEPENDENCE AVE, SW
WASHINGTON DC 20591

AFIWC/MSY 1
102 HALL BLVD, STE 315
SAN ANTONIO TX 78243-7016

ATTN: KAROLA M. YOURISON 1
SOFTWARE ENGINEERING INSTITUTE
4500 FIFTH AVENUE
PITTSBURGH PA 15213

USAF/AIR FORCE RESEARCH LABORATORY 1
AFRL/VSOSA(LIBRARY-BLDG 1103)
5 WRIGHT DRIVE
HANSCOM AFB MA 01731-3004

ATTN: EILEEN LADUKE/D460 1
MITRE CORPORATION
202 BURLINGTON RD
BEDFORD MA 01730

OUSD(P)/DTSA/DUTD 1
ATTN: PATRICK G. SULLIVAN, JR.
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202

MR ROBERT GRAYBILL 1
DARPA/ITO
3701 N FAIRFAX DR
ARLINGTON VA 22203-1714

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.